Journal of Software Engineering and Applications, 2016, 9, 250-290
Published Online May 2016 in SciRes. http://www.scirp.org/journal/jsea
http://dx.doi.org/10.4236/jsea.2016.95019

Scientific Research Publishing

An Approach to Parallel Simulation of Ordinary Differential Equations

Joshua D. Carl[1], Gautam Biswas[2]

[1]Department of Electrical Engineering and Computer Science, Milwaukee School of Engineering, Milwaukee, WI, USA
[2]Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, USA
Email: carl@msoe.edu, gautam.biswas@vanderbilt.edu

Received 19 March 2016; accepted 28 May 2016; published 31 May 2016

Abstract

Cyber-physical systems (CPS) represent a class of complex engineered systems where functionality and behavior emerge through the interaction between the computational and physical domains. Simulation provides design engineers with quick and accurate feedback on the behaviors generated by their designs. However, as systems become more complex, simulating their behaviors becomes computation all complex. But, most modern simulation environments still execute on a single thread, which does not take advantage of the processing power available on modern multi-core CPUs. This paper investigates methods to partition and simulate differential equation-based models of cyber-physical systems using multiple threads on multi-core CPUs that can share data across threads. We describe model partitioning methods using fixed step and variable step numerical integration methods that consider the multi-layer cache structure of these CPUs to avoid simulation performance degradation due to cache conflicts. We study the effectiveness of each parallel simulation algorithm by calculating the relative speedup compared to a serial simulation applied to a series of large electric circuit models. We also develop a series of guidelines for maximizing performance when developing parallel simulation software intended for use on multi-core CPUs.

Keywords

Parallel and Multi-Thread Programming, Ordinary Differential Equations, Simulation

1. Introduction

Cyber-physical systems represent a class of complex engineered systems whose functionality and behavior emerge through the interaction between the computation and physical domains; in addition, these interactions

can occur, locally, within a system, or be distributed in a networked environment [1]. CPSs are also frequently designed to include human decision making as part of the control of the physical system.

Systems engineering methods play an important role in designing and analyzing CPSs. In the traditional approach to systems engineering, design has followed a discipline by discipline approach, with individual components being created in isolation to meet specified design criteria. After the individual, compartmentalized, design phases, all of the components are brought together for integration testing to verify that the design criteria are met [2]. This method works well for simpler systems with few interacting components and physical domains.

With the increasing prevalence and complexity of present-day CPSs, the traditional systems engineering approach is proving to be detrimental to the overall design process. Several research papers, such as [1]-[5], have discussed in detail the problems and challenges involved in designing and building CPSs. For CPSs, to analyze and understand the behaviors of the system requires methods by which the different component models can be composed and analyzed both as a full system and as individual components. One approach is to build all of the components in a virtual design environment, and use simulation to perform integration testing throughout the design process [4]. At the beginning of the design the different components can be represented by low fidelity models, possibly taken from a component library, and composed to form an initial design of the final system. As the design progresses the initial models can be replaced with more specific and detailed models, including the final component implementations. At all points in the design process the composed system can be simulated, providing a means to analyze how the integrated system performs [4]. Simulation, therefore, provides a very tight design-test-redesign feedback loop for the designer. Any changes or tweaks to the model can be made quickly and simply in the virtual design environment and the test re-run to verify for correctness.

Previously, the increasing complexity of systems and the corresponding increases in their computational complexity were matched by faster processor speeds that kept the simulation runtime within reasonable bounds. However, since the year 2005 processor clock speeds have largely leveled off (see **Figure 1**), and the increase in computing power for commercial chips has been achieved by adding processor cores that can execute in parallel rather than by increasing clock speed [6]. Exploiting this parallel processing power provided by multi-core architectures to improve the run time of a simulation requires algorithmic changes to the simulation; one has to develop parallel versions of the simulation algorithms to speed up the computation. However, developing these parallel simulation algorithms will require careful consideration of the physical CPU architecture to derive the best parallel performance.

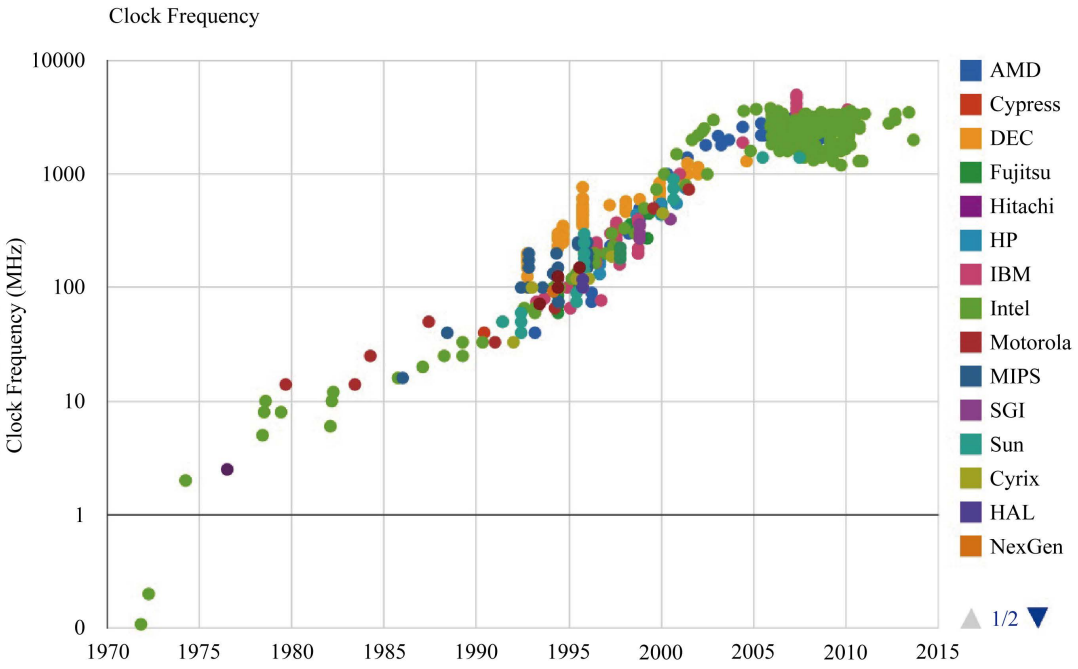

Figure 1. Processor clock frequency vs. time [6].

Modern CPUs have several layers of memory between the physical CPU registers and the main system memory collectively called the cache (the cache is discussed in more detail in Section 3.1.2). The cache allows CPU core to keep data that it is currently working on readily available in a layer of cache that provides very quick access, and allows data that is not needed to stay in a layer of cache farther away from the core [7]. The cache also allows the physical computation cores to communicate with each other. Mismanagement or ignoring the structure of the CPU cache in a multi threaded program can have a drastic negative impact on the program execution performance [7]-[9], and therefore, parallel software design on a multi-core CPU requires careful consideration of both how to partition the computational problem into independent pieces suitable for parallel simulation, and how those independent pieces interact with each other and the on chip memory.

The potential parallel processing power of modern multi-core CPUs, coupled with the potential performance pitfalls of the CPU cache, leads us to focus our research into parallel simulation algorithms that target a multi-core CPU and use the CPU cache as a performance asset instead of a liability.

The goal of this paper is to take steps toward facilitating the design process for cyber-physical systems by reducing the time it takes to simulate complex and large system models by developing parallel simulation algorithms for multi-core CPU architectures. A primary component of these algorithms is the incorporation of suitable memory management and the use of program constructs that take advantage of the CPU memory architecture. Our research contributions, therefore, focus on:

1) Developing classes of parallel simulation algorithms that appropriately uses the multiple cores and the cache memory organization on a multi-core CPU, and

2) Running experimental studies that help us analyze the effectiveness of various multi-threading and memory management schemes for parallel simulations.

The contents of this paper are as follows. Section 2 describes our simulation methodology. Section 3 describes our shared memory parallel processing architecture. Section 4 our approach to parallel simulation of ordinary differential equations and the models that we use to evaluate our algorithms. Section 5 presents our parallel simulation algorithms and experiment results. Section 7 presents our conclusions regarding parallel simulation of ordinary differential equations.

2. Simulation Methodology

This section provides definitions and derivations related to simulation (Section 2.1), describes our approach to simulation (Section 2.2), the numerical integration methods we use to perform our simulations (Section 2.3), and an overview of inline integration (Section 2.4).

2.1. Definitions and Discrete Time Derivation

The physical systems we will be working with are continuous dynamic systems, and are modeled using ordinary differential equations (ODE) or differential algebraic equations (DAE). ODE models are represented mathematically in the state equation form

$$\dot{\mathbf{x}}(t) = \mathbf{f}_x\left(t, \mathbf{x}(t), \mathbf{w}(t)\right). \tag{1}$$

where $\mathbf{x}(t)$ is a vector of the state variables of the system, with a corresponding derivative vector $\dot{\mathbf{x}}(t)$. $\mathbf{w}(t)$ is the set of algebraic variables, which includes the input variables. n_x is the number of state variables, and n_w is the number of algebraic variables [10]. The vectors $\mathbf{x}(t)$ and $\dot{\mathbf{x}}(t)$ have the same dimensions equal to n_x. The functional form $\mathbf{f}_x$ specifies the values of the derivatives of each state variable, which when integrated provides the evolution of the system behavior, $\mathbf{x}_t$ over time. The continuous time ODE set, $\mathbf{f}_x\left(t, \mathbf{x}(t), \mathbf{w}(t)\right)$, can be expressed in discrete-time form by plugging Equation (1) into a numerical integration method, the Forward Euler method from Equation (3) in this case [11]:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}_n\left(t_n, \mathbf{x}_n, \mathbf{w}_n\right) \tag{2}$$

All of the parameters have the same meaning as the previous equation, except they are in terms of the current simulation step *n* instead of continuous time. It is possible that the discrete time function, $\mathbf{f}_n$ will be very different from the continuous time function $\mathbf{f}_x$. However, if the time interval between time steps *n* and $n+1$,

represented by *h*, is small, the first order approximation above is quite accurate.

One important point to note from Equation (1) is that the calculation of any value in $\dot{\mathbf{x}}(t)$ does not depend on any other values of $\dot{\mathbf{x}}(t)$ because no value of $\dot{\mathbf{x}}(t)$ appears as inputs to the function $\mathbf{f}_x$. This means that the calculation of the specific values in $\dot{\mathbf{x}}(t)$ can happen in any order.

Solving the functions in $\mathbf{f}_x$ is called a function evaluation.

Definition 1 (Function Evaluation) *A function evaluation is one evaluation of the vector function* $\mathbf{f}$ *from Equation* (1).

There is one equation for each variable that is not a state variable, that is, one equation for each element in the sets $\dot{\mathbf{x}}(t)$ and $\boldsymbol{w}(t)$, which makes the system just determined. At time point, t_i, the state variable derivatives, $\dot{\mathbf{x}}(t_i)$ are integrated using a numerical integration method to calculate the values of the state variables at time $\mathbf{x}(t_{i+1})$. This interaction between $\dot{\mathbf{x}}(t)$ and $\mathbf{x}(t)$ relate to the structure of a simulation, and is reviewed in Section 2.2. Integration methods are reviewed in Section 2.3.

2.2. Simulation Structure

The goal of a simulation is to generate time trajectory data for all of the variables in the dynamic system model. The high-level process is shown in **Figure 2**. The simulation is performed in discrete time, so the relevant variables are functions of the simulation step, $\dot{\mathbf{x}}_n$, instead of time, $\mathbf{x}(t)$. Each value of *n* represents a specific point in time, *t*. The value of *t* at the next time step, $n+1$, is $t+h$, where *h* is the time step of the simulation, which the amount of time between time steps.

In modern simulation software the model compiler converts the declarative DAE model to explicit ODE form [12]. The ODE set of equations calculate the values of the derivatives of the state variables, $\dot{\mathbf{x}}_n$ (Equation (1)). After the state variable derivatives are calculated, the simulation software passes the calculated derivatives to the chosen solver.

Definition 2 (Numerical Integration Method) *The numerical integration method integrates the derivative of a variable at time t to determine the value of the corresponding state variable at time* $t+h$*, where h is the simulation step size.*

Definition 3 (Step Size) *The step size of a simulation is the distance between two individual time steps, and, depending on the solver, it may be kept fixed or dynamically adjusted during simulation. It is described by the variable h in definition* 2.

Definition 4 (Solver) *The solver is a standalone piece of software that implements a specific numerical integration method, and, potentially, step size control, order control, and any other tasks necessary to accurately complete a simulation* [13].

The solver uses a numerical integration method to integrate the derivatives to find the new values of the state variables at the next time step, $\mathbf{x}_{n+1}$, or in continuous time representation $\mathbf{x}(t+h)$. The values are checked to guarantee that the results are within the prescribed error tolerance. If they are not then the solver takes an appropriate action, usually halving the simulation step size, and tells the simulation code to re-evaluate the previous time step with the smaller step size. If the state variables are within the defined tolerance, the new state variables are passed to the computational model which then calculates the next values of the state variable derivatives. The solver can also take the step of making the simulation step size larger if a small step size is no longer needed. This process continues until the simulation stop time is reached.

The two aspects of the simulation, shown in **Figure 2**, the computational model and the solver are generally two separate pieces of software. This allows different integration algorithms to be paired with the same model by simply changing a setting in the overall modeling environment.

2.3. Integration Method Overview

The numerical integration methods that we use in this work are the Forward Euler (FE), and Runge-Kutta-Fehlburg 4-5 (RKF45). The FE method is defined as [11]:

$$x_{n+1} = x_n + h \cdot \dot{x}_n. \tag{3}$$

The Runge-Kutta family of methods [14] are multi-stage methods, where the method calculates the values of *x* and $\dot{x}$ at micro-steps (stages) between *t* and $t+h$. The equations for the Runge-Kutta-Fehlburg 4-5 method are:

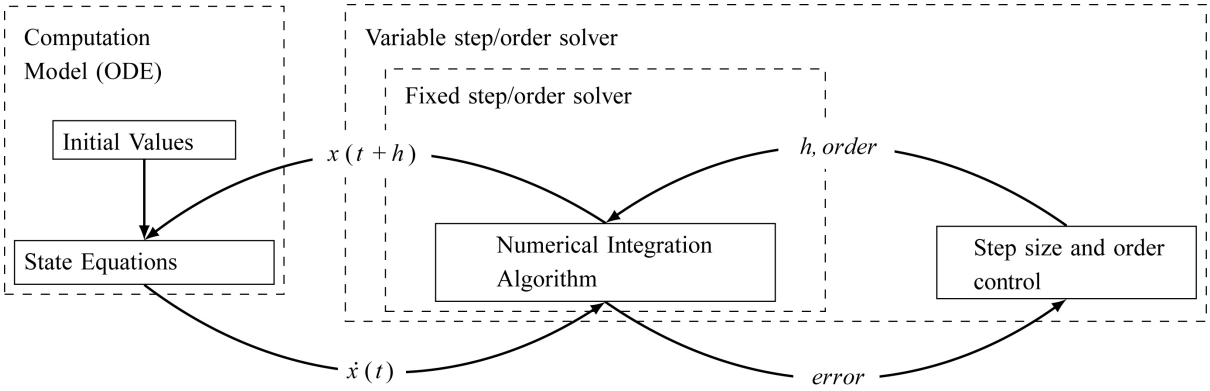

Figure 2. Simulation structure.

$$1^{st}\ \text{stage}: k_1 = f\left(x_n, t_n\right)$$

$$2^{nd}\ \text{stage}: k_2 = f\left(x_n + \frac{h}{4}\cdot k_1, t_n + \frac{h}{4}\right)$$

$$3^{rd}\ \text{stage}: k_3 = f\left(x_n + \frac{h\cdot 3}{32}\cdot k_1 + \frac{h\cdot 9}{32}, t_n + \frac{h\cdot 3}{8}\right)$$

$$4^{th}\ \text{stage}: k_4 = f\left(x_n + \frac{h\cdot 1932}{2197}\cdot k_1 - \frac{h\cdot 7200}{2197}\cdot k_2 + \frac{h\cdot 7296}{2197}\cdot k_3, t_n + \frac{h\cdot 12}{13}\right)$$

$$5^{th}\ \text{stage}: k_5 = f\left(x_n + \frac{h\cdot 439}{216}\cdot k_1 - 8\cdot h\cdot k_2 + \frac{h\cdot 3680}{513}\cdot k_3 - \frac{h\cdot 845}{4104}\cdot k_4, t_n + h\right)$$

$$6^{th}\ \text{stage}: k_6 = f\left(x_n - \frac{h\cdot 8}{27}\cdot k_1 + 2\cdot h\cdot k_2 - \frac{h\cdot 3544}{2565}\cdot k_3 - \frac{h\cdot 1859}{4104}\cdot k_4 - \frac{h\cdot 11}{40}, t_n + \frac{h}{2}\right)$$

$$\text{Final 1}: x_{1\_n+1} = x_n + h\cdot\left[\frac{25}{216}\cdot k_1 + \frac{1408}{2565}\cdot k_3 + \frac{2197}{4104}\cdot k_4 - \frac{1}{5}\cdot k_5\right]$$

$$\text{Final 2}: x_{2\_n+1} = x_n + h\cdot\left[\frac{16}{135}\cdot k_1 + \frac{6656}{12825}\cdot k_3 + \frac{28561}{56430}\cdot k_4 - \frac{9}{50}\cdot k_5 + \frac{2}{55}\cdot k_6\right]. \quad (4)$$

This method is really 2 separate RK methods that share their initial stages. One method is a 4^{th} order method that uses stages 1 through 5 and equation Final 1, and the second method is a 5^{th} order method that uses stages 1 through 6 and uses equation Final 2. This makes determining the error for the simulation trivial. The error for a time step is evaluated according to:

$$\left|x_{1\_n+1} - x_{2\_n+1}\right| < \epsilon. \quad (5)$$

If Equation (5) is false then the simulation needs to choose a new step size and recalculate the time step. If Equation (5) is true then the time step is accepted by the solver and the 5^{th} order value of x_{n+1} is propagated to the next time step. If the difference calculation in Equation (5) is below both the error threshold and a separate step size threshold, then the solver may choose to make the time step greater.

2.4. Inline Integration

Inline integration was first introduced in [15] and is described in some detail in [12]. Inline integration works by removing the separation between the model equations and the integration method. The integration equations are inserted ("inlined") directly into the model equations, and **Figure 2** can be modified into **Figure 3** when inline

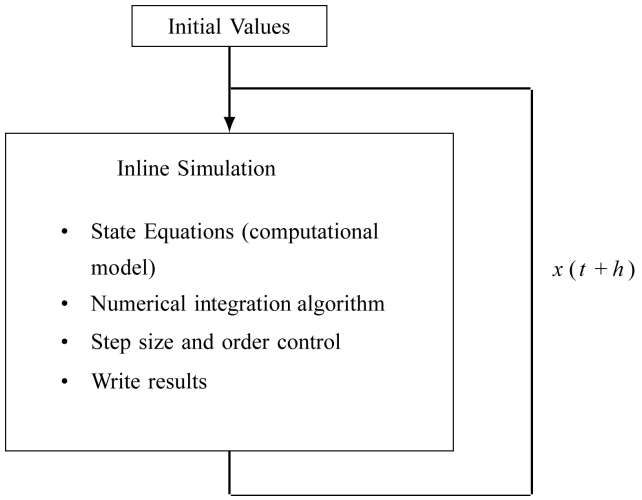

Figure 3. Inline integration simulation structure.

integration is used. By itself, inline integration will not necessarily yield a reduction in simulation time, but it will provide a means for parallelizing the integration method with the model, and open up other opportunities for reduce the simulation time.

3. Shared Memory Parallel Processing Architecture

Traditionally computing architectures have been described in terms of four qualitative categories according to Flynn's taxonomy [16] [17]:

1) Single Instruction, Single Data (SISD): This architecture is a conventional sequential computer with a single processing element that has access to a single program data storage.

2) Multiple Instruction, Single Data (MISD): In this architecture there are multiple processing elements that have access to a single global data memory. Each processing element obtains the same data element from memory, and performs its own instruction on the data. This is a very restrictive architecture, and no commercial parallel computer of this type has been built [17].

3) Single Instruction, Multiple Data (SIMD): In this architecture, multiple processing elements execute the same instruction on their own data element. Applications with a high degree of parallelism, such as multimedia and computer graphics, can be efficiently computed using a SIMD architecture [17]. Modern tools for parallel processing large amounts of database information, such as Hadoop [18], are closely related to the SIMD architecture.

4) Multiple Instruction, Multiple Data (MIMD): In this architecture, there are multiple processing elements and each element accesses and performs operations on its own data. The data may be accessed from local memory on the processor, from shared memory, *i.e.*, memory shared by multiple processor units.

3.1. Multi-Core CPU

Multi-core CPUs, such as Intel's Core line of processors [19] and AMD's FX processors [20], are the backbone of modern engineering workstations and are an example of MIMD parallel architecture. We are targeting improved performance of simulation on engineering workstations so the MIMD architecture will be our focus in this research.

At a conceptual level, the CPU can be divided into two parts: the computational processor, and the memory. Our primary focus will be on the processor memory, which is covered in Section 3.1.2, but we will first summarize the important aspects of the processor as it relates to the memory system.

3.1.1. Processor

Current high end chips have several processor cores [19] [20], and each processor core can handle two computational threads through hardware multithreading.

Definition 5 (Thread) *A single program execution stream that includes the program counter, the register state and the stack* [21]. *Multiple threads can share an address space, meaning threads can access memory used by other threads. Due to this shared address space, a processor can switch between threads without invoking the operating system.*

Definition 6 (Hardware Multithreading) *Increasing utilization of a processor by switching to another thread when one thread is stalled due to causes such as waiting for data from memory, or a no operation instruction* [21].

Definition 7 (Process) *A higher level computation unit, above threads. A process includes one or more threads, the address space, and the operating system state. Changing between processes requires invoking the operating system* [21].

CPU hardware that implements hardware multithreading is designed such that each hardware thread is presented to the Operating System (OS) as an individual CPU, which gives the OS double the number of cores to use for computational work. In our work, we treat these multithreaded processors as individual processors, just like the OS, so that a computer with 4 cores that supports 8 threads (2 threads per core) is treated as having 8 unique cores.

3.1.2. Cache

The cache memory hierarchy is the memory closest to each processor core. The cache closest to each core is called the L1 cache, and, in a 3 level hierarchy such as Intel's Core architecture [19], the cache farthest away from the core is called L3. Beyond the L3 cache is the main system memory (RAM). An example of the Intel Core i7 cache structure is in **Figure 4**. In the Intel i7 the L1 cache is split into an instruction cache and a data cache, and each L1 and each L2 cache are usable by only 1 core, while the L3 cache is shared across all cores. This L3 cache allows the individual cores to exchange data, and the L1 cache is used to keep frequently used data readily available, as cache access times become longer the farther away the level of cache is from the core. Also, the cache levels that are farther away from a processor are going to be larger than the levels closer to the processor. On the CPU we use to perform our experiments in Section 5 each L1 cache is 32 KB, each L2 cache is 256 KB, and the L3 cache is 8 MB [19].

In most cases, the cache is implemented as a hierarchy, where data cannot be in the L1 cache without it also being in L2 and L3 [22]. More formally:

$$\text{if } x = \text{some data} \tag{6}$$

$$x \in L1 \Rightarrow x \in L2 \Rightarrow x \in L3. \tag{7}$$

In order for processors to communicate or exchange data, the data needs to be in the layer(s) of cache that are shared across cores. Usually, as shown in **Figure 4**, this is the cache that is the farthest from the processor, and

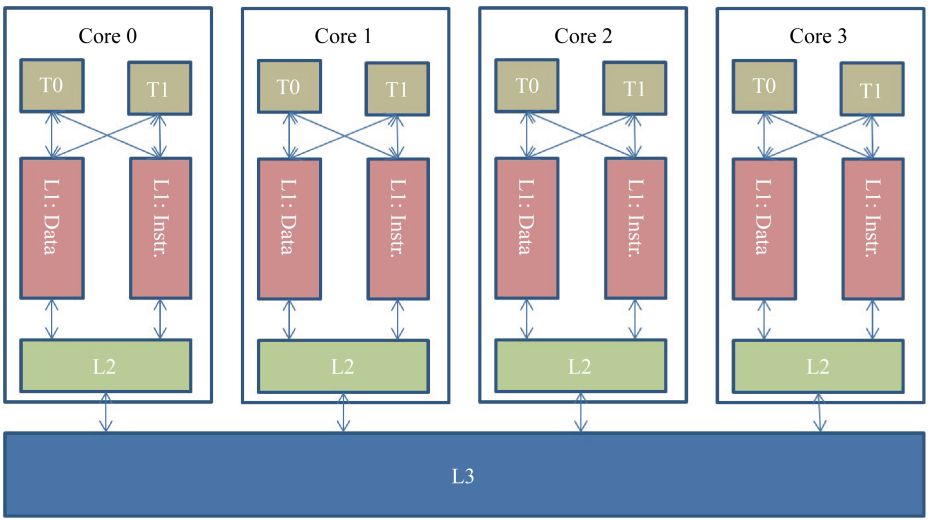

Figure 4. An example of the memory structure of an Intel i7 processor [9] [19].

therefore the slowest layer of cache. As an example, since the threads can only directly access their L1 cache, to send data from Core 0 to Core 2, Core 0 will first have to write its data to the L1 cache. Then Core 2 will request the data through a load or get instruction. The memory controller on the chip will then move the data from the L1 cache on Core 0, to the L2 cache on that core, and then to the global shared L3 cache. After it is in the L3 cache the data will be moved into the Core 2 L2 cache, and finally into the L1 cache on Core 0 where the data can finally be used.

The cache is organized into cache lines, or blocks.

Definition 8 (Cache Line) *The minimum unit of information that can be present or not present in a cache* [22].

Definition 9 (Cache Line Read) *The process of pulling a cache line from a cache far away from a CPU core to a cache close to the CPU core.*

In most modern desktop processors the cache line is 64 bytes. This means that moving 4 bytes of data (typically, the size of an integer in C/C++) to the L1 cache will move the entire 64 byte cache line that contains the requested 4 bytes of data. Multiple threads in different cores accessing data in the same cache line can lead to a problem known as cache line sharing.

Definition 10 (Cache Line Sharing) *When two unrelated variables are located in the same cache line are being written to by threads on separate cores, the full line is exchanged between the two cores even though the cores are accessing different variables* [22].

Cache line sharing can have a dramatic impact on performance, which can be demonstrated through a simple experiment (modified from [8]). We create a large array of data, and a varying number of threads, from 1 up to n, where n is equal to 2 times the number of processors recognized by the operating system. We assign each thread a location in the array on which to perform some work. Since arrays are stored in contiguous memory, if we assign threads 1 through n indexes in array from 0 through $n-1$, the threads will all be working with data on the same cache line. If we assign each thread an index in the array of 0 through $(n-1) * $ variables in a cache line then each thread will be working with data on separate cache lines. The expectation is that the execution times when the threads are working on the same cache line will be much longer than the execution times when the threads are working on separate cache lines.

This experiment was performed on a machine running Ubuntu 15.04 with an Intel i7-880 CPU clocked at 3.08 GHz and 8 GB of RAM. This CPU has 4 physical cores that each support 2 hardware threads, the same architecture as **Figure 4**. The experiment was run 20 times and the computation times averaged. The results of the experiment are shown in **Figure 5**. The results show that when a cache line is shared across threads, as the number of threads increases the computation time increases as well; nearly linearly in terms of the number of threads. The results also show that when the threads each have their own cache line, there is very little increase

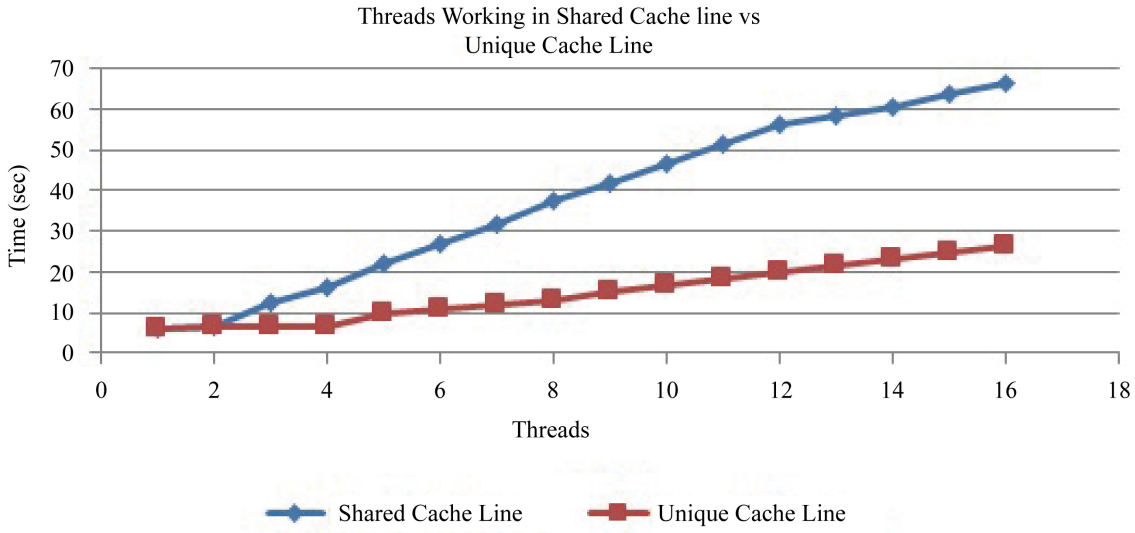

Figure 5. Results of an experiment comparing with multiple threads working on data that exists in the same cache line to multiple threads working on data that exists in separate cache lines.

in the computation time when using 1 through 4 threads. This is expected as the threads are working on independent cache lines and do not interfere with each other. The jump in computation time above 4 threads is likely due to hardware limitations of Intel's hardware multithreading implementation (marketed as Hyperthreading by Intel [19]). There may appear to be 8 unique CPUs to the operating system, but the physical hardware cannot accurately mimic 8 independent cores when there are only 4 physical cores to work with. When using 4 threads, the method using independent cache lines is 2.9 times faster than the implementation using a shared cache line.

As this example shows, sharing cache lines between threads can lead to significant performance problems, and should be avoided [8] [21]. One method to avoid cache line sharing is to use cache aligned data structures.

Definition 11 (Cache Aligned Data) *Cache Aligned Data is data that has a memory address that is an even divisor of the cache line size. That is, in most modern architectures, cache aligned data has memory address modulo* 64 = 0.

In effect, a cache aligned data structure was used in the above example where each thread was assigned an array index in which to work that was on its own cache line. Programmers need to consider the processor cache, and avoid cache line sharing, when designing multithreaded applications to ensure program performance.

4. Parallel Simulation of Ordinary Differential Equations

This section presents our general approach to simulating our models (Section 4.1), a mathematical description of model partitioning (Section 4.2), a discussion on the overhead associated with running parallel simulations (Section 4.3), the models that we used to evaluate our algorithms (Sections 4.4 and 4.5), our experiment configuration (Section 4.6), and our base test case (Section 4.7).

4.1. Model Simulation

The state equations can be integrated independently and in any order for each time step (Section 2.1). This gives us freedom, within a time step, to group the state equations in ways that achieve the best run time performance. At the end of a time step, the updated state variable values will need to be synchronized across the state equations of the system, to allow the different execution threads to acquire the updated state variable values before starting the calculations for the next time step.

We implemented two types of integrators for the parallel simulation algorithms: 1) a fixed-step Euler integrator, and 2) a variable-step Runge-Kutta integrator. We discuss our implementations for the two integration schemes next.

4.1.1. Fixed Step Integration

We used an Inline Forward Euler (IFE) integration method [11] (Equation (3)) for all of our fixed step simulations. Each step of a Forward Euler (FE) integrator in discrete time is described as:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \dot{\mathbf{x}}_n, \tag{8}$$

where h is the step size. Inlining the integration equation allows our simulation to calculate $\mathbf{x}_{n+1}$, in a single equation instead of two equations one calculating the derivative of the state variable, $\dot{\mathbf{x}}_n$, and then a second calculating the value of the state variable at $\mathbf{x}_{n+1}$, as is traditionally done in simulation (see Section 2.2). In this case, inline integration works by taking the equation for ODEs, Equation (1), which solves for $\dot{\mathbf{x}}(n)$ and substituting that equation into the equation for FE integration, Equation (8), *i.e.*,

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h \cdot \mathbf{f}_x\left(t_n, \mathbf{x}_n, \mathbf{w}_n\right). \tag{9}$$

We simplify the definition of this equation to be:

$$\mathbf{x}_{n+1} = \mathbf{f}_{IFE}\left(h, \mathbf{x}_n, \mathbf{f}_x\left(t_n, \mathbf{x}_n, \mathbf{w}_n\right)\right) \tag{10}$$

During simulation using the IFE method, for each time step the simulation calculates the values for $\mathbf{x}_{n+1}$ during time step n. After all of the values for $\mathbf{x}_{n+1}$ are calculated the simulation copies $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$:

$$\mathbf{x}_n = \mathbf{x}_{n+1}. \tag{11}$$

and then advances the simulation time:

$$t_n = t_n + h, \tag{12}$$

where t is the simulation time and h is the time step. Then the simulation loops and repeats the process until the simulation reaches the simulation end time. In our fixed step simulations the step size of the simulation is passed as a parameter to the simulation at run-time. The simulation algorithm using IFE integration is described in **Algorithm 1**.

4.1.2. Variable Step Integration

We used an explicit Runge-Kutta-Fehlberg 4,5 (RKF4,5) solver (described in Equation (4)) from the GNU Scientific Library (GSL) [23] for our variable step simulations. The GSL implements the RKF4,5 as a standalone solver that follows a traditional approach to simulation, as described in Section 2.2 and shown in **Figure 2**. To use the solver we supply a function for calculating the derivatives of the state variables, Equation (1), and a function for calculating the system Jacobian matrix. The solver determines the time step size, performs the integration, and maintains the synchronization between time steps. The simulation algorithm using the RKF4,5 solver is described in **Algorithm 2**. Since this is a variable step solver, we supply an output interval that indicates how frequently we want to receive updates to the system state.

4.2. Mathematical Description of Model Partitioning

Partitioning any computational problem into independent components, such that each part can be executed on a separate processor, is the first step to solving the problem in parallel [24]. In our work we rely on simple heuristic partitioning schemes, and leave the problem of finding an optimal model partitioning to future work. Even determining simple heuristic schemes has been a non-trivial problem, but our work represents good progress in that direction. The results derived here can influence the algorithms we develop in the future for optimal partitioning. Another part to this work, is to investigate the number of parallel threads that produce optimal run-time performance, this means that in our experiments we will vary the number of model partitions we create.

To facilitate the partitioning, it is important to note from Equation (1) is that the calculation of any value in $\dot{\mathbf{x}}(t)$ does not depend on any other values of $\dot{\mathbf{x}}(t)$, because no value of $\dot{\mathbf{x}}(t)$ appears as inputs to the function $\mathbf{f}_x$. This means that the calculation of the specific values in $\dot{\mathbf{x}}(t)$ can happen in any order. This fact allows us to divide the functions represented by $\mathbf{f}_x$ into subsets, where each subset contains one or more of the equations to compute the derivatives, and is assigned to one of the execution threads on a multi-core CPU. These threads execute in parallel on the CPU, but at the end of each simulation time step the threads will have to synchronize their data to preserve the correctness of the simulation. This approach allows us to parallelize the computation of each time step in the simulation, and should reduce the wall clock-time of the simulation.

Therefore, we take the dynamic system model described in ODE form (Equation (1)), and divide the system into ℓ sets, where ℓ is the number of threads being used. In this work, we partition the equations such that an

```
1 while t_n < StopSimulationTime do
2 |   x_{n+1} = f_IFE(h, x_n, f_x(t_n, x_n, w_n)) ;
3 |   x_n = x_{n+1} ;
4 |   t_n = t_n + h ;
5 end
```

Algorithm 1. Algorithm describing simulation using Inline Forward Euler integration.

```
1 current_time = 0;
2 output_time = output_interval ;
3 while current_time < StopSimulationTime do
4 |   RKF45_Integrate(f_x, from = current_time, to = output_time, RKF45 output = x_n) ;
5 |   current_time = output_time ;
6 |   output_time = output_time + output_interval ;
7 end
```

Algorithm 2. Algorithm describing simulation using a variable step RKF4,5 solver.

approximately equal number of state variable calculations are assigned to each thread[1].

$$\dot{\mathbf{x}}_n = \dot{\mathbf{x}}_{n\_0} + \dot{\mathbf{x}}_{n\_1} + \cdots + \dot{\mathbf{x}}_{n\_\ell} \tag{13}$$

$$\mathbf{x}_n = \mathbf{x}_{n\_0} + \mathbf{x}_{n\_1} + \cdots + \mathbf{x}_{n\_\ell} \tag{14}$$

$$\mathbf{x}_{n+1} = \mathbf{x}_{(n+1)\_0} + \mathbf{x}_{(n+1)\_1} + \cdots + \mathbf{x}_{(n+1)\_\ell} \tag{15}$$

$$\begin{aligned}\mathbf{f}\left(t, \mathbf{x}_n, \mathbf{w}_n\right) = \mathbf{f}_0\left(t, \mathbf{x}_{n\_0}, \cdots, \mathbf{x}_{n\_\ell}, \mathbf{w}_n\right) \\ + \mathbf{f}_1\left(t, \mathbf{x}_{n\_0}, \cdots, \mathbf{x}_{n\_\ell} + \mathbf{w}_n\right), \cdots, \mathbf{f}_\ell\left(t, \mathbf{x}_{n\_0}, \cdots, \mathbf{x}_{n\_\ell}, \mathbf{w}_n\right).\end{aligned} \tag{16}$$

Differences in the cardinality of the sets accounts for remainders in integer division of n_x/ℓ. Subsets $1, \cdots, (n_x \text{ modulo } \ell)$, will have a cardinality of $n_x/\ell + 1$, and subsets $(n_x \text{ modulo } \ell + 1), \cdots, \ell$ will have a cardinality of n_x/ℓ.

The value of ℓ is dependent on the simulation algorithm. For example, if $\ell = n_x$, there is one state variable per subset in each partition. For all of the other tests ℓ is going to have a value of m_{test}, where m_{test} represents the number of threads being used for a test and ranges from 1 to *m*, where *m* is the number of parallel threads on the CPU.

As discussed earlier, this partitioning approach allows us to complete the calculations within each time step in parallel, but it will require a synchronization phase at the end of each time step to guarantee correct results. The partitioning, and the memory structure that supports the chosen partition, will differ for each of our parallel simulation algorithms, and the specific differences are highlighted in Section 5, which describes the parallel simulation algorithms and the results of the experimental runs with those algorithms.

4.3. Overhead in Running Parallel Simulations

In any parallel program the parallelization adds overhead that is not present in a serial execution of a program. The number of computations per time step of the simulation is the same for sequential and parallel algorithms. However, the overhead generated by parallelization can limit the effectiveness of the parallel implementation. This overhead typically takes the form of scheduling overhead and communication overhead. The created threads need to be assigned, by the operating system, to a CPU core during the thread runtime. It is the responsibility of the OS to ensure that all threads are given enough time on a CPU core to complete their work. There is also overhead due to communication between the created threads. This communication overhead does not occur in a single threaded program, but it is necessary in a parallel program synchronization. The run time of a parallel program can be described at a high level as:

$$\text{wall clock time} = \text{computation time} + \text{overhead}. \tag{17}$$

The challenge in designing parallel software is to minimize the overhead component of Equation (17). This will generally involve two components: 1) the overhead required to manage shared memory between the different execution threads, and 2) the amount of swapping that needs to be performed when there are more threads than available processor cores. These two components are not independent of each other, and we describe a number of algorithms that trade off these two parameters.

4.4. RLC Simulation Models

We use the Modelica modeling language [25] to create our simulation models. Modelica is a programming language for describing the behavior of physical systems [25]. At its core, it is an acausal, object oriented, and equation based language for describing physical models using differential, algebraic, and discrete equations. The language supports hierarchy, model inheritance, and traditional programming language structures such as if-statements, case statements, loops, built-in primitive variable types, and parameters.

The RLC circuit models were implemented to exploit Modelica's hierarchical nature, which allows for efficient construction of models from individual components. The base component is shown in **Figure 6**. The inductor and capacitor implementations are from the Modelica Standard Library (MSL) [25], version 3.2. The

[1]This partitioning may not work very well when the structure of the differential equations is not homogeneous.

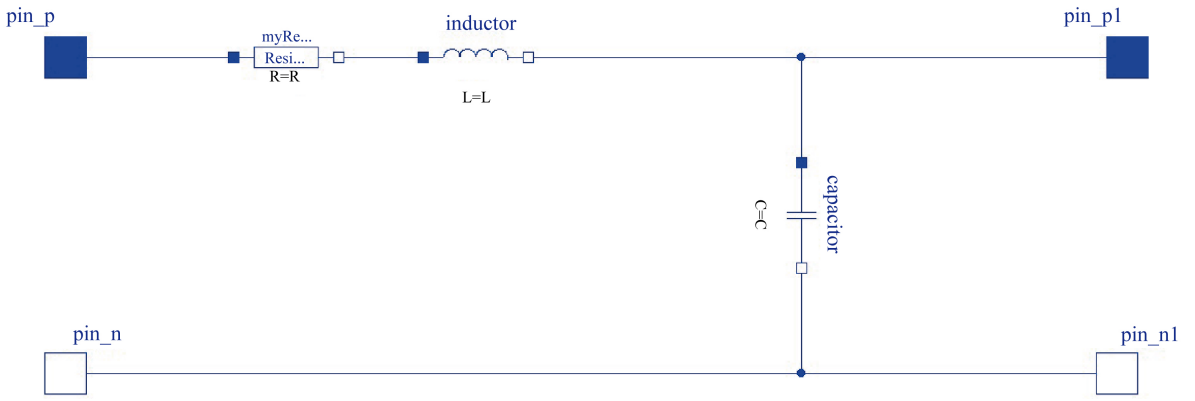

Figure 6. Base RLC component model with 2 state variables.

resistor is a modified version of the MSL resistor model. The thermal components were removed from the resistor model because they contain Modelica if statements, which can lead to hybrid behavior. The terminals associated with the components are also from the MSL and were not altered. This set of base components were used to create a larger component with six connected base components as shown in **Figure 7**. This larger component was used to create the models we used for our experiments. We created a model with 288 state variables that used 24 of the components shown in **Figure 8**, and we created a model with 804 state variables that used 67 components as shown in **Figure 9**.

4.5. RLC Models with Algebraic Loops

Algebraic loops are very common in modern complex models [11]. To evaluate the performance of parallelizing the simulation of a model with algebraic loops, we modified the models in **Figure 8**, and **Figure 9** to include components that introduced algebraic loops. We followed the same approach to building the models with algebraic loops as we did for the models described above: we used Modelica's hierarchical structure to build complex components from the base components. We created 2 new base components, one with a linear algebraic loop and one with a non-linear algebraic loop. They are shown in **Figure 10** and **Figure 11**, respectively. The algebraic loop is introduced due to the series resistors ([26] also has an example of series resistors causing algebraic loops), and, as a result, in these circuit fragments there is no way to uniquely determine the voltage drop across each resistor. Therefore, the voltage drop across the resistors must be calculated simultaneously because the voltage across each resistor depends on the voltage across every other resistor. The non-linear loop is due to one of the resistance values in the loop being a non-linear function of the voltage across another resistor in the loop.

To create models with algebraic loops, we modified the models in **Figure 8** and **Figure 9** by replacing some of the components in **Figure 7** with the components in **Figure 10** and **Figure 11** to create the component in **Figure 12**. These new models are shown in **Figure 13** and **Figure 14**. The loop components we added were split approximately evenly between the linear loop components and non-linear loop components.

4.6. Experiment Configuration

We used two sets of parameter values in our experiments. The first set of values set all parameters equal to 1, and the resulting electrical circuits have slow time constants. A parameter value of 1 for all an electrical components of a circuit is not realistic for most applications, but it was useful for an initial evaluation of a parallel algorithm. We also used more realistic parameter values of 15 Ω for all resistors, 15 mH for all inductors, and 250 μF for all capacitors. The circuits using these parameters had much faster time constants.

We will measure the simulation run time using the wall-clock time of the simulation.

Definition 12 (Wall-Clock Time) *The wall-clock time of a simulation is the time it takes from an external user's perspective for the overall simulation to be completed, i.e. the time for a simulation to complete as measured by a clock on a wall.*

We use the wall-clock time so that we can study the reduction in simulation time using parallel algorithms

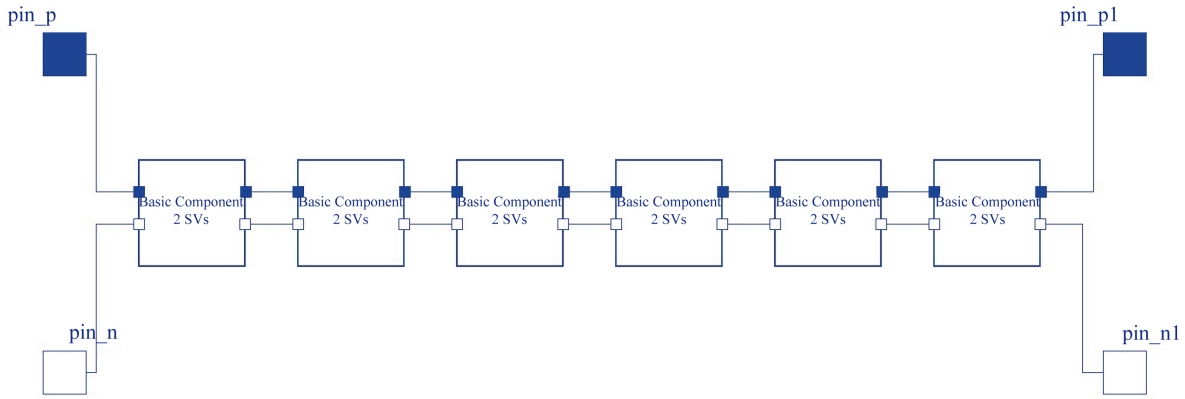

Figure 7. RLC component model with 12 state variables.

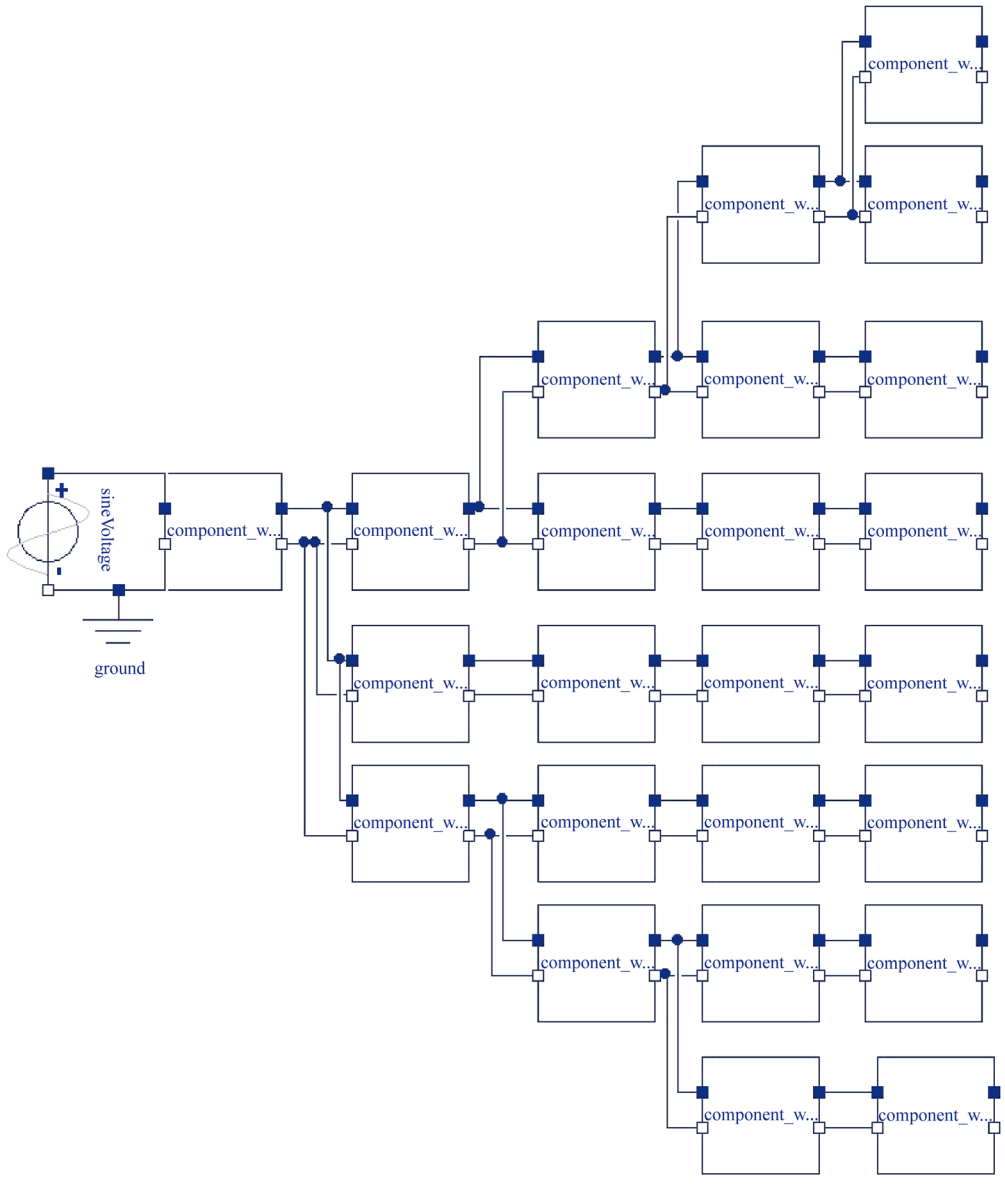

Figure 8. Complex RLC model with 288 state variables.

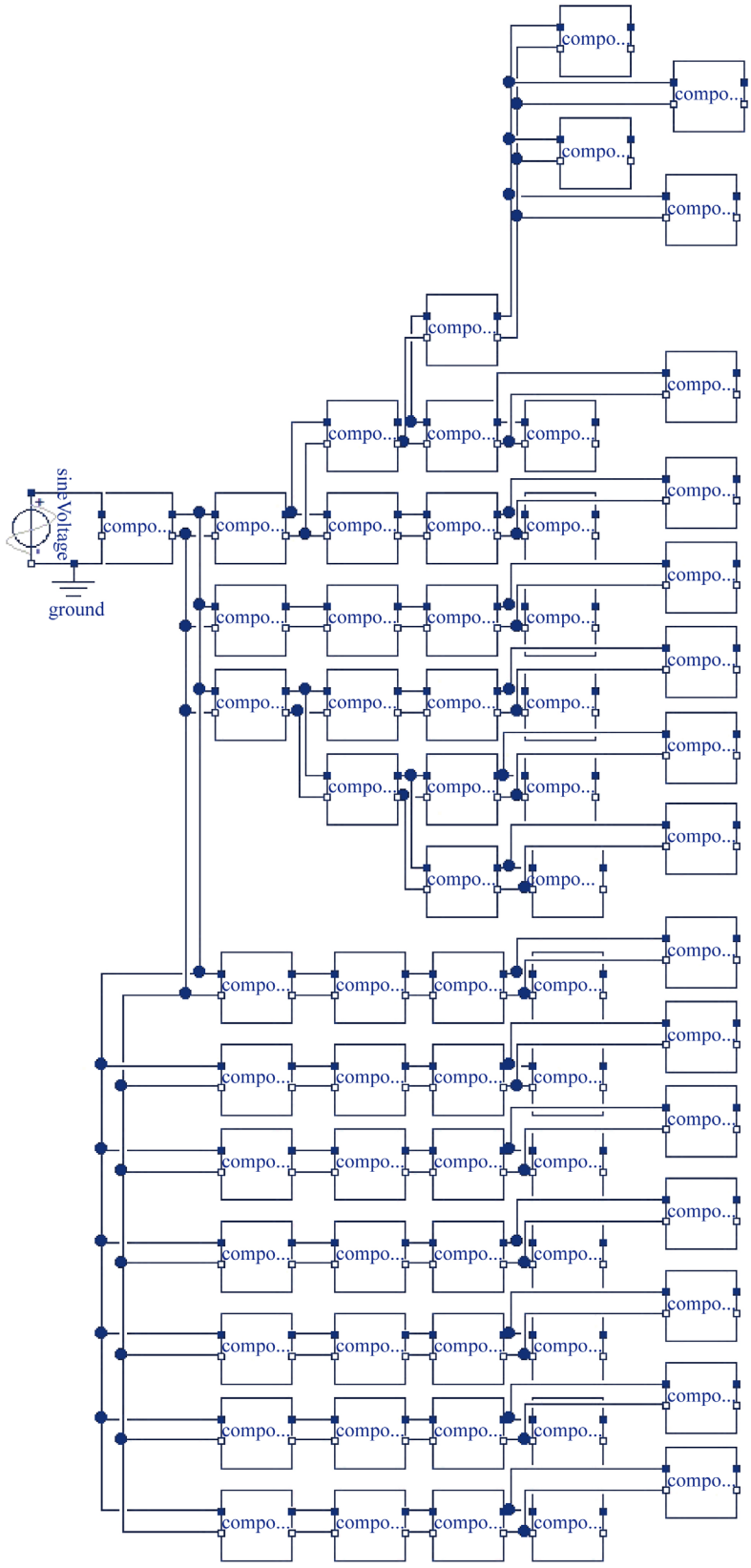

Figure 9. Complex RLC model with 804 state variables.

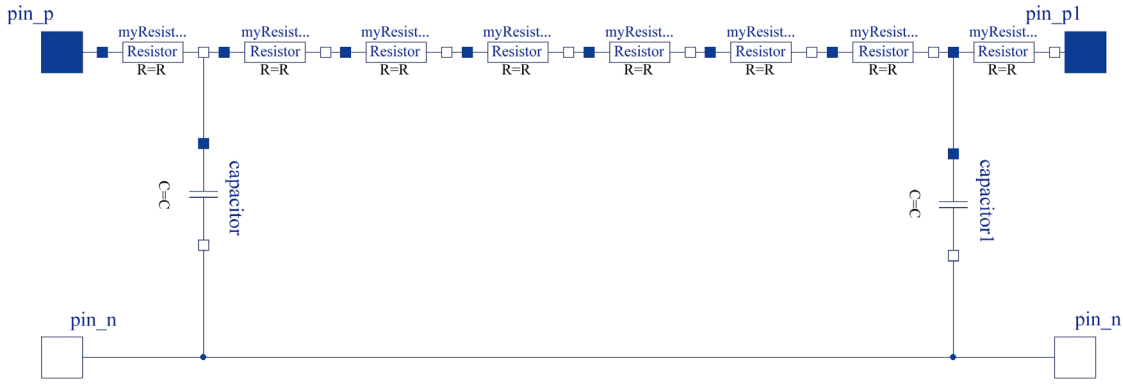

Figure 10. Base RLC component model with 2 state variables and a linear loop.

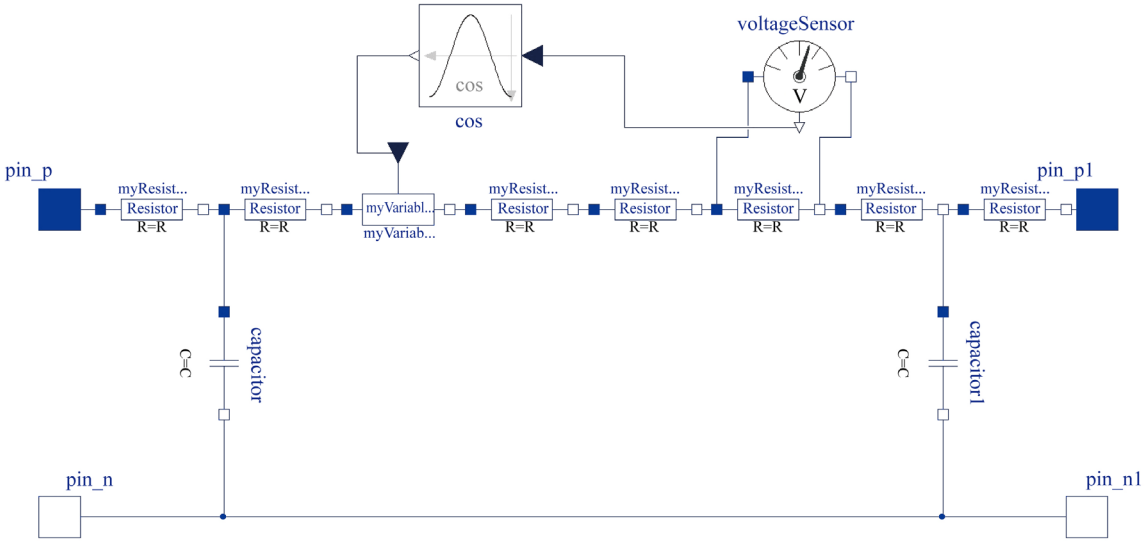

Figure 11. Base RLC component model with 2 state variables and a non-linear loop.

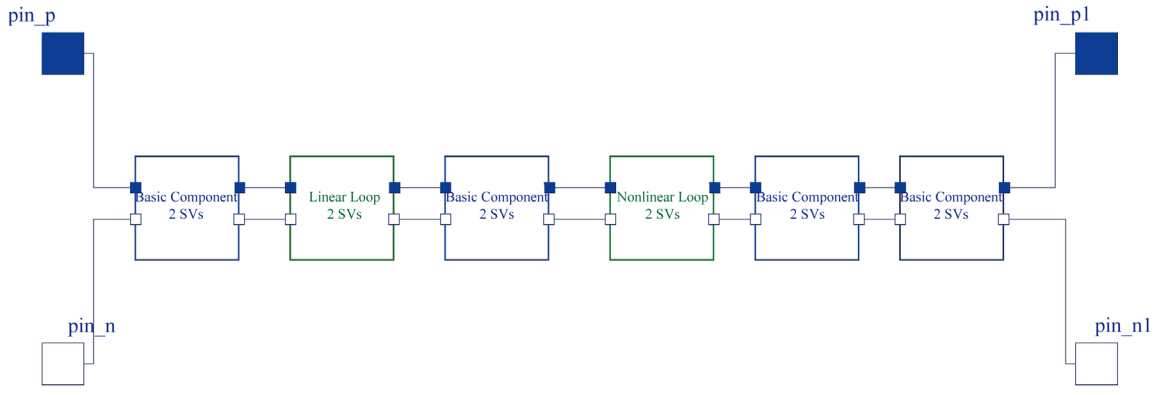

Figure 12. RLC component model with 12 state variables and 2 algebraic loop components.

from the user's point of view. In a parallel program it is also possible to measure time as the aggregate of the busy time of each CPU, but this is not useful from an end user perspective. It is a better measure of the performance from a hardware use perspective. We measure the effectiveness of a parallel simulation as the relative speedup of the wall clock time for a parallel simulation compared to a serial simulation. The equation to calculate the relative speedup is:

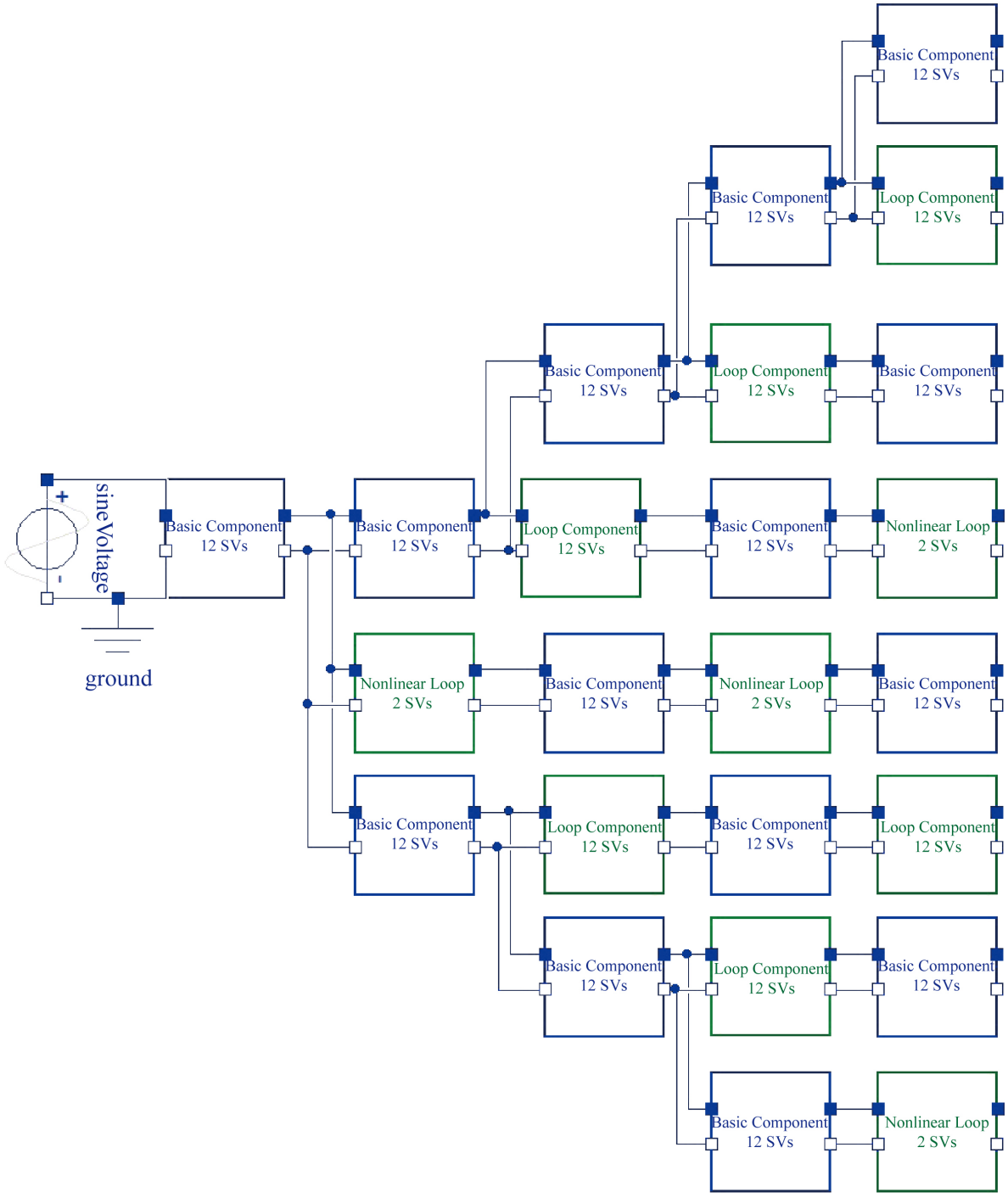

Figure 13. Complex RLC model with 288 state variables and 19 algebraic loops (9 linear and 10 non-linear).

$$\text{rel. speedup} = \frac{t_{serial}}{t_{parallel}}. \tag{18}$$

A value of greater than 1 implies that the parallel simulation provides a speedup, and a value of less than 1 indicates that the parallel code runs slower than the serial code. The simulation times, t_{serial} and $t_{parallel}$ are the measured wall-clock times for the two simulations.

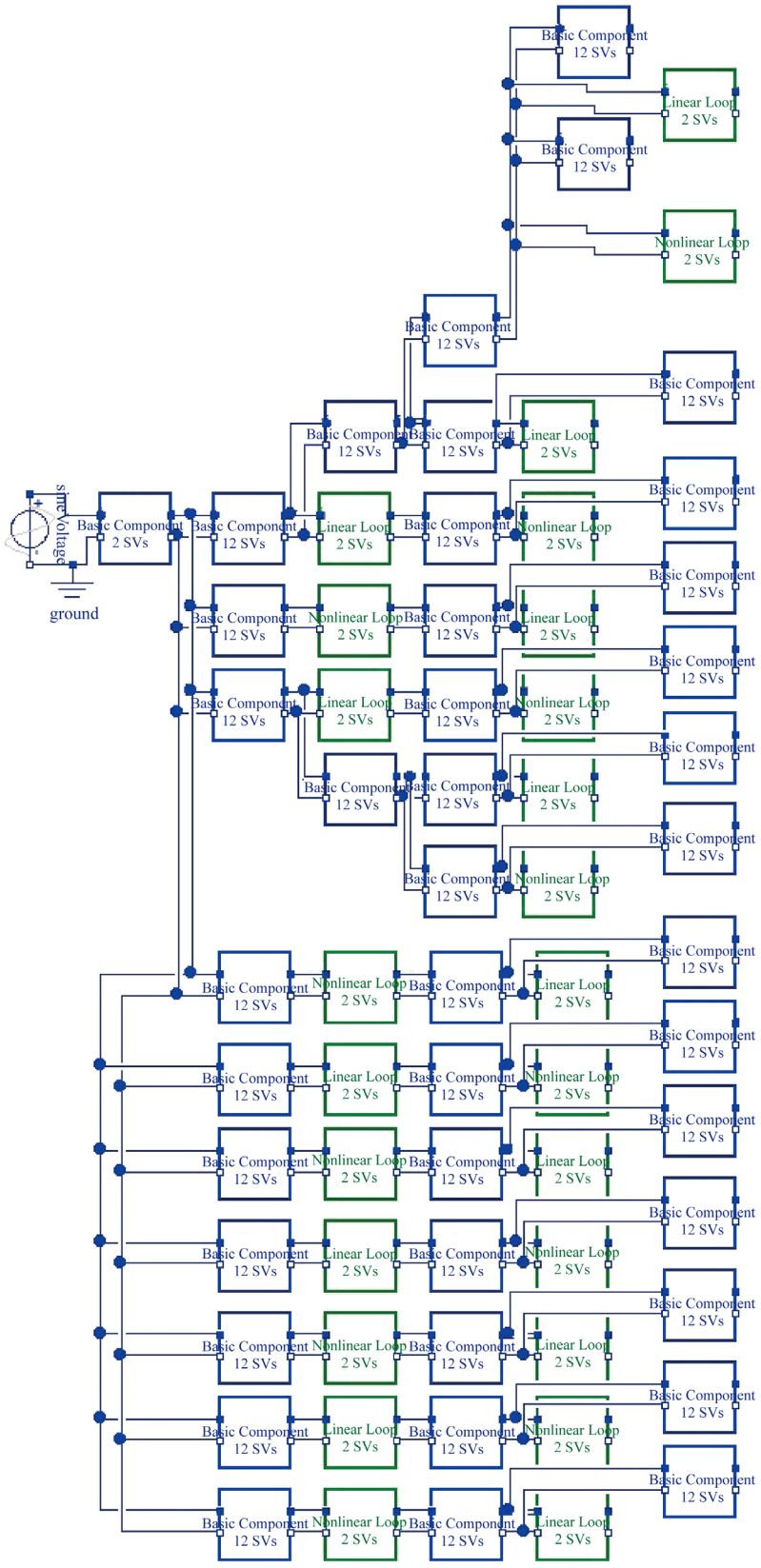

Figure 14. Complex RLC model with 804 state variables and 26 algebraic loops (14 linear and 12 non-linear).

The simulation programming language used for this study is C/C++, compiled and run on Linux. C/C++ simplifies the programming task and generates very efficient execution code. Further, C/C++ also has low-level memory management functions that allow cache-aligned data structures to be created. We target Linux as a simulation environment because it provides more control over the created threads, and generally faster execution than Windows.

Unless otherwise noted, all experiments were run on an Intel i7-880 desktop PC clocked to 3.08 GHz with 8 GB of memory running Ubuntu 15.04. The generated C++ code was compiled using g++ version 4.9.2 [27]. All experiments were run 20 times on the models presented in Sections 4.4 and 4.5 for parameter values that created slow and fast behavioral time constants. The experiments were run without writing state variable time trajectory data to disk in order to guarantee more consistent simulation times. To calculate the relative speedup for a particular algorithm and thread count, we calculate the average of the wall clock time for the serial simulation. Using this average we then calculate the relative speedup for each algorithm and thread count. The relative speedups are averaged for a thread count, and the standard deviation is calculated from the relative speedups.

4.7. Base Test Case

We focus on comparing our parallel simulation algorithms to the fastest and most basic serial simulation algorithm we were able to create. Comparing a parallel implementation to a fast serial implementation of the same problem was advocated by [28] as a means to prove that the parallel implementation provides a practical benefit, and is not simply an academic exercise. The type of serial simulation, either fixed step or variable step, is matched to the type of parallel simulation we are using, so fixed step parallel simulations are compared to a fixed step serial simulation, and variable step parallel simulations are compared to a variable step serial simulation.

5. Parallel Simulation Algorithms

A parallel simulation algorithm that appropriately uses multiple cores and the CPU cache has conflicting goals, even though using multiple cores effectively and managing the cache both focus on the physical hardware of the CPU. An ideal program architecture from the standpoint of a multi-core CPU will involve a limited number of parallel computation threads, with very little sharing of data between the threads. An ideal program architecture from the standpoint of the CPU cache might be to divide the program into many very small pieces so that the data being worked on by each thread fits entirely into the size of one cache line. These two ideal architectures are often in conflict with each other, one prefers large partitions and the other small partitions, and finding the right balance between the two is the key aspect of our research.

Partitioning the system of equations, such that the cache can be used effectively to minimize the communication between the execution threads was a key factor that drove the design of the simulation algorithms presented in this section. We focus on 1) Minimizing the communication between execution threads, and 2) Utilizing the fastest communication methods for exchanges that have to take place. We leverage the shared-memory architecture in modern multi-core CPUs so that communication between threads can be handled in hardware as a part of the processor's cache (parallel architectures and processor cache are covered in Section 3.1). However, we design our algorithms so that they share only a minimum amount of data, as sharing data through the cache across threads also causes computation delays, as we demonstrated in Section 3.1.2.

This section presents the set of parallel simulation algorithms we developed in a progression. The different simulation algorithms varied on how many threads were created, and how the variables in the sets $\boldsymbol{x}_{n+1}$ and $\boldsymbol{x}_n$ were divided into blocks of memory corresponding to the created threads. We also paid special attention to minimizing memory sharing across threads, in an effort to maximize the performance of the CPU cache, and by extension the simulation speedup. The differences in memory sharing are a primary differentiator between our simulation algorithms, and a primary driver in reworking the thread assignments.

This section describes definitions related to the simulation threads (Section 5.1), a brief description of the initial algorithms we developed that did not produce good results (Section 5.2), and a detailed description of the algorithms that did produce results (Sections 5.3, 5.4 and 5.5).

5.1. Thread Definitions

The set of threads used in each experiment is:

$$\mathbf{T} = \{T_0, T_1, T_2, \cdots, T_\ell, T_{main}\}, \tag{19}$$

where $\mathbf{T}$ is the set of all threads in the experiment, T_0 through T_ℓ are the threads created for parallel execution of the particular simulation algorithm, and T_{main} is the primary thread that spawns the child threads and controls the advance of the simulation time steps. The value of ℓ typically ranges between 0 and $m-1$, where m is the number of CPUs available to the operating system (on a processor that implements hardware multithreading that number of CPUs available to the OS is going to be double the number of cores on the processor, see Section 3.1). As an extreme we also developed an algorithm where ℓ was set to the number of state Equations (n_x) of the dynamic system model. We also define:

$$\mathbf{T}_{spawn} \subset \mathbf{T} = \{T_0, T_1, T_2, \cdots, T_\ell\} \tag{20}$$

to identify the threads that were created by the main thread, T_{main}, for the simulation. Each algorithm creates one or more memory blocks that will be assigned to the threads. We describe the size of the memory blocks as:

$$M[X], \tag{21}$$

where M represents a block of memory, and X is the size of that memory. We will also use the symbols $\rightarrow$, $\leftarrow$, and $\leftrightarrow$ to describe if a thread writes to a memory block, reads from a memory block, or reads and writes to a memory block, respectively. Each of the threads in $\mathbf{T}_{spawn}$ only communicates its status to T_{main} and does not have to communicate with any other thread.

Another factor that drove the implementation of our algorithms was to enable fast communication and simple synchronization between hardware threads, *i.e.* $\mathbf{T}_{spawn}$ and T_{main}. Synchronization is handled using shared variables. Simple spin locks are used at synchronization points to pause threads [29]. In a spin lock, a thread continually polls a variable waiting for it to change value, which provides for fast communication to handle synchronization between threads because all communication is handled in hardware through the processor's cache. As soon as the assigned variable is updated in the waiting thread's cache, a thread can continue its computation. Alternatives to spin locks are semaphore or mutex locking, but these locking schemes allow the operating system to move the waiting thread off of the core and replace it with a separate waiting thread. Restarting the original thread will require not only monitoring the semaphore or mutex in question, but also requires the OS to move the thread back onto the core. Whereas this may not be significantly impact computational efficiency for some applications, it can significantly deter the execution time of the parallel simulation algorithms, which require synchronization once or twice a time step. Since the simulation runs complete many time steps (up to 50,000 in some experiments) the extra work by the OS to put the thread back into active processing causes significant negative impact on the simulation. Spin locks, because the thread stays active, do not suffer from this problem.

5.2. Initial Algorithms

Our initial attempts to produce a speedup did not produce good results. This section summarizes our initial algorithms, and the lessons learned from them that were applied to our later algorithms.

5.2.1. Algorithm 1: One Thread per State Variable

Our first parallel algorithm creates a separate thread for each individual function in $\mathbf{f}_{IFE}$ (*i.e.*, state equation + inline code for performing an integration step). Therefore:

$$\mathbf{T}_{spawn} = \{T_0, T_1, \cdots, T_{n_x}\}.$$

Even for moderately sized models, this results in many more threads than cores available on a typical multicore CPU (at the time of this writing, typically there are 4 to 8 cores available [19] [20]). This algorithm creates one shared memory block across all threads defined by:

$$M[2 \cdot n_x],$$

where M represents the block of memory, n_x represents the number of state variables in the system, and the size of the memory block in terms of the number of variables in the block is $2 \cdot n_x$. The memory block is twice

the number of state variables because there are two variables for each state variable: one to store the current time step value of the state variable ($\mathbf{x}_n$), and one to store the next time step value ($\mathbf{x}_{n+1}$). Each of the threads has read/write access to the single shared memory block:

$$\mathbf{T} \leftrightarrow M. \tag{22}$$

We also tested a second version of this algorithm that set the thread affinity for each of the created threads, such that the threads were evenly distributed between the processor cores. The expectation with setting the thread affinity is that it would offload the work of dynamically scheduling the threads from the OS and fix the schedule, thus reducing the simulation time.

Definition 13 (Thread Affinity) *The thread affinity identifies on which processor a thread is allowed to run.*

5.2.2. Algorithm 2: Full Shared Memory with Agglomeration

Our second parallel algorithm agglomerated our state variable calculations so that we could create fewer than n_x threads, where n_x is the number of state variables in the model. **Algorithm 2** partitions $\mathbf{f}_{IFE}$ into subsets with each subset executing in parallel on a separate thread. Since each thread contains multiple state equations, each thread requires larger amounts of computational work, and this creates a better balance between the computational work and overhead required for scheduling and communication. We developed two different agglomeration strategies that we call (1) simple agglomeration and (2) smart agglomeration.

Algorithm Full Shared Memory Simple Agglomeration, uses a simple agglomeration scheme where the equations in $\mathbf{f}_{IFE}$ are partitioned according to the order they were specified in the original model. This means that the first $n_x/(m-1)$ equations were assigned to T_0, the second set of $n_x/(m-1)$ equations were assigned to T_1, and so on until all of the equations in $\mathbf{f}_{IFE}$ were assigned to the threads in $\mathbf{T}_{spawn}$. There may be variations in the sizes of the subsets as described in Section 4.2.

Algorithm Full Shared Memory Smart Agglomeration, uses a smart agglomeration scheme that groups the ODE equations, such that the equations that have a large number of dependencies (used a large number of state variables to calculate a particular value of $\mathbf{x}_{n+1}$) were grouped together in the same thread in $\mathbf{T}_{spawn}$. The idea behind smart agglomeration is to limit the amount of data communication between threads, which reduces the overhead, and, therefore, should reduce the simulation time.

5.2.3. Algorithm 3: Partial Partitioned Memory

This parallel simulation algorithm, Partial Partitioned Memory, is identical to the simple agglomeration approach presented above, except that it partitions $\mathbf{x}_{n+1}$ into $m-1$ independent memory blocks that match the subsets of $\mathbf{f}_{IFE}$. This avoids a single large shared memory block across all threads, and the created memory blocks can be represented as:

$$M_0\left[\frac{n_x}{m-1}\right], M_1\left[\frac{n_x}{m-1}\right], \cdots, M_{m-2}\left[\frac{n_x}{m-1}\right], \tag{23}$$

where each memory block M_x is aligned to a cache line, and each block is writable by only one of the threads in $\mathbf{T}_{spawn}$. There is also a shared memory block

$$M_{main}\left[n_x\right] \tag{24}$$

that contains the values of $\mathbf{x}_n$ and is readable by all of the threads in $\mathbf{T}_{spawn}$, but is only writable by thread T_{main}. Since thread T_{main} has the same role as **Algorithm 1** and **Algorithm 2**, and it is responsible for synchronizing $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$, it needs to write to M_{main} and read from memory blocks $M_0, \cdots, M_{m-2}$.

5.2.4. Analysis of Early Algorithms

None of our initial algorithms produced a speedup compared to the serial case. Our first algorithm produced speedups on the order of 10^{-3} in the best case, which means that the parallel algorithm was orders of magnitude slower than the serial algorithm. Our second algorithm produced speedups of 0.92 in the best case, which means that our parallel algorithm was almost as fast as the serial algorithm. Our third algorithm produced speedups of 0.49 in the best case, which means that our parallel algorithm was half as fast as a serial algorithm. We derived a number of lessons from these first parallel algorithms that we applied to our later algorithms. These lessons in-

clude:

1) Do not create more threads than there are processors,

2) Managing thread workload so that the computational work assigned to each thread is greater than the overhead associated with creating the thread,

3) Avoid cache line sharing between computational threads,

4) Evenly divide computational work between all threads, and do not reserve one thread for controlling the simulation and all other threads for computation, and

5) Reduce the communication and dependencies between threads.

The algorithms that did produce a speedup are discussed in detail in the following sections.

5.3. Full Partitioned Memory Parallel Fixed-Step ODE Simulation

The fixed-step simulations that gave us the best performance used fully distributed memory. This means that each thread has its own cache aligned block of memory that it writes to which includes both $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$. We tested two different versions of the algorithm, Full Partitioned Memory Minimum Sharing and Full Partitioned Memory Simple Agglomeration.

These two algorithms expand the roles of the threads in $\mathbf{T}_{spawn}$ and the thread T_{main}. The threads in $\mathbf{T}_{spawn}$ now perform their own synchronization step when given a signal by T_{main}. Thread T_{main}, instead of remaining idle when computations are being completed by threads in $\mathbf{T}_{spawn}$, solves some of the functions in $\mathbf{f}_{IFE}$. The program flow describing these two partitioned memory algorithms is described in **Figure 15**. Compared to the previous algorithms, the back and forth flow between threads is eliminated. Since each of the threads are responsible for their own synchronization according to Equation (11), the synchronization and advancing simulation time steps, on lines 3 and 4 of **Algorithm 1**, are performed in parallel. The threads in $\mathbf{T}_{spawn}$ pause before synchronizing to make sure all threads have completed calculating their values of $\mathbf{x}_{n+1}$. They proceed to the synchronization phase of the simulation on a signal from T_{main}. The threads again wait after synchronization for a signal from T_{main} before they begin to calculate their assigned equations from $\mathbf{f}_{IFE}$. This second synchronization point guarantees that there are no race conditions between the threads in $\mathbf{T}$ during synchronization. Even though each thread calculates its own simulation time variable, only thread T_{main} can issue a stop signal to the threads in $\mathbf{T}_{spawn}$ when the simulation is complete.

5.3.1. Full Partitioned Memory Minimum Sharing

The first full distributed memory approach, Full Partitioned Memory Minimum Sharing, created memory blocks:

$$\begin{aligned} &M_0\left[2\cdot\left(n_x/m\right)\right], M_1\left[2\cdot\left(n_x/m\right)\right], \cdots, \\ &M_{m-2}\left[2\cdot\left(n_x/m\right)\right], M_{shared}\left[2\cdot\left(n_x/m\right)\right]. \end{aligned} \tag{25}$$

Each memory block is created on its own cache line to avoid cache line sharing. Also, it is assigned a subset of variables from $\mathbf{x}_{n+1}$ and $\mathbf{x}_n$, and then assigned to a thread in $\mathbf{T}_{spawn}$. The block of memory M_{shared} is assigned to T_{main}. This is shown in **Figure 16**. The memory blocks are structured so that each thread in $\mathbf{T}_{spawn}$ only needs the variables in its memory block and the variables in M_{shared} to solve its subset of $\mathbf{f}_{IFE}$. Therefore, each thread in $\mathbf{T}_{spawn}$ only writes its assigned $\mathbf{x}_{n+1}$ values to its own block of memory, and only reads from M_{shared}. Thread T_{main} has control of M_{shared}, and the variables stored in M_{shared} are the variables that are needed in more than one thread. To calculate its values of $\mathbf{x}_{n+1}$, T_{main} has read access to all of the other memory blocks. This memory structure is called Minimum Sharing, because each of the threads in $\mathbf{T}_{spawn}$ only share memory with thread T_{main}.

Aligning the data structures to a cache line boundary is accomplished through the align as C++ keyword introduced in the C++11 standard [30]. The align as keyword is used in the declaration of a variable or other object, and takes a parameter that describes the memory alignment requirement for the variable being declared. As an example the code align as (64) double var1; will create a new double called var1, and the memory address of var1 will be an even multiple of 64. Since our processor uses a 64 byte cache line, using align as (64) when a variable is declared will align that variable to a cache line. The keyword applies when creating arrays and structs as well, and will provide a means to create an individual memory blocks for each thread in $\mathbf{T}$ that is aligned to

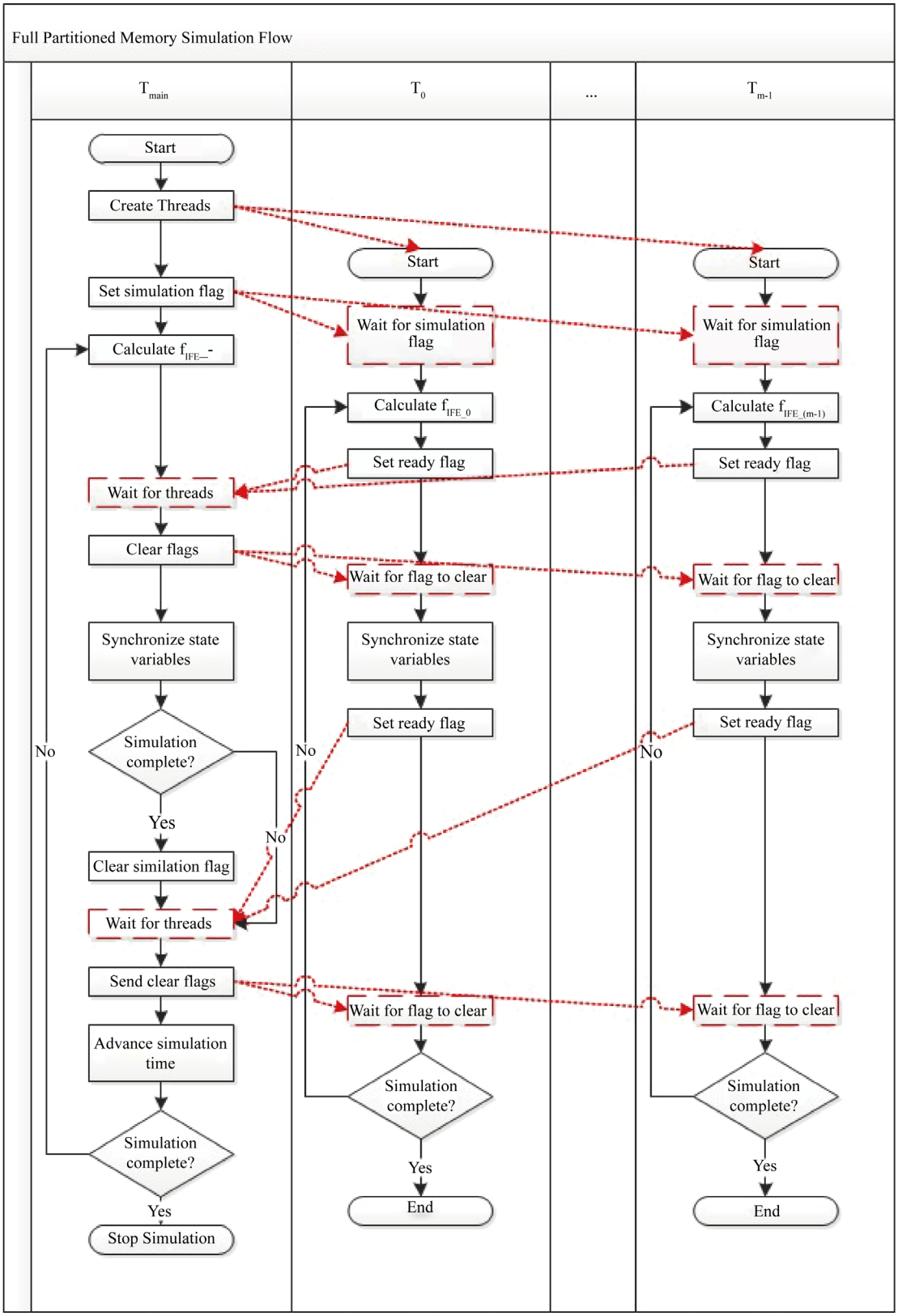

Figure 15. Full parallel simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

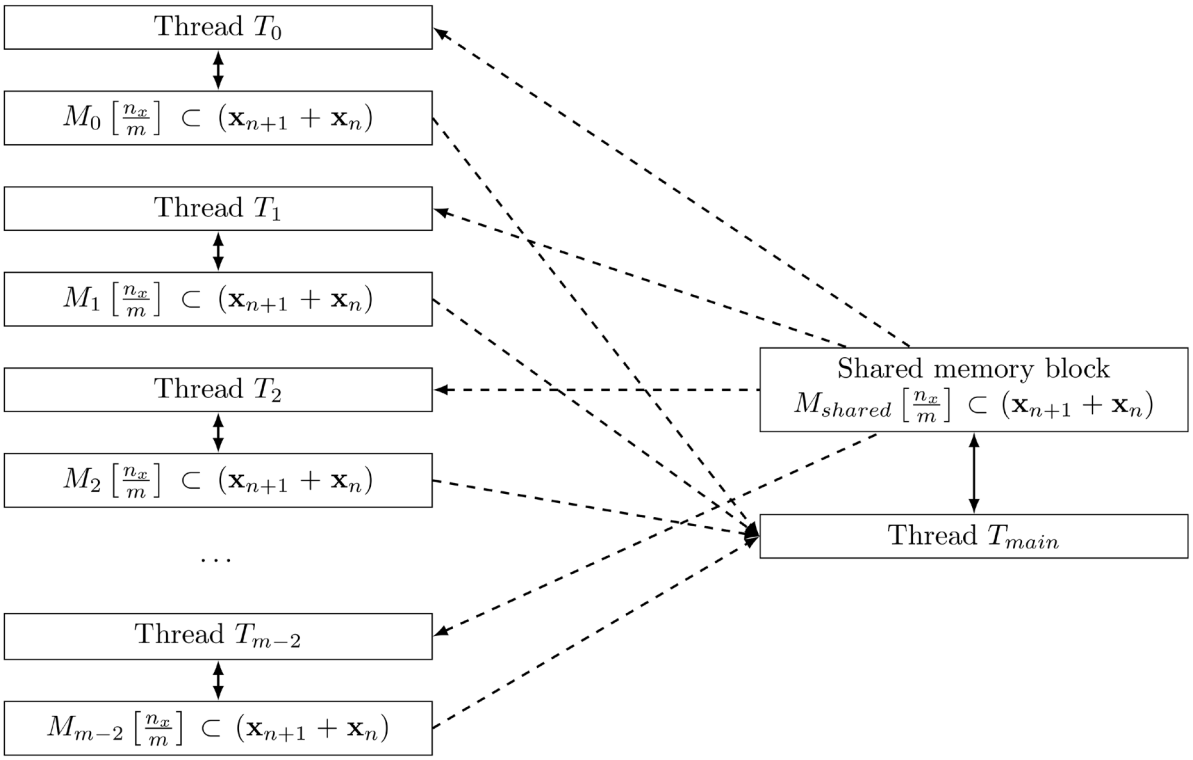

Figure 16. Figure describing Full Partitioned Memory Minimum Sharing memory structure using 1 thread per CPU core. The dashed lines indicate a read-only relationship.

a cache line. If each thread only writes to one memory block, and all memory blocks are aligned to separate cache lines, then there will be no cache line sharing between threads.

5.3.2. Full Partitioned Memory Simple Agglomeration

The second full distributed memory approach, Full Partitioned Memory Simple Agglomeration. It has the same program flow as Full Partitioned Memory Minimum Sharing, shown in **Figure 15**, where thread T_{main} solves a subset of $\mathbf{f}_{IFE}$ but still controls when to terminate the simulation, and the threads in $\mathbf{T}_{spawn}$ synchronize their assigned variables.

The memory approach used in Simple Agglomeration is a little different from Minimum Sharing. Each thread in $\mathbf{T}$ has its own local memory block to which it reads and writes, and that represents the subset of values in $\mathbf{x}_{n+1}$ for which the thread is responsible. There also is a series of shared memory blocks $M_{shared0}, M_{shared1}, \cdots, M_{sharedm-1}$, where each block is a subset of $\mathbf{x}_n$. All of the threads in $\mathbf{T}$ can read from all of the shared blocks, but only one thread writes to each shared block. The portion of M_{shared} that each thread writes to is aligned to a cache line boundary, so there are no cache line sharing problems. The memory model of Full Partitioned Memory Simple Agglomeration is:

$$\begin{aligned} &M_0\left[(n_x/m)\right], M_1\left[(n_x/m)\right], \cdots, \\ &M_{m-1}\left[n_x/m\right], M_{shared0}\left[n_x/m\right], M_{shared1}\left[n_x/m\right], \cdots, M_{sharedm-1}\left[n_x/m\right]. \end{aligned} \tag{26}$$

The relationship between the threads and the different memory blocks is shown in **Figure 17**. With this approach, each of the threads in $\mathbf{T}$ needs read access to M_{shared}, but only writes to its own local block and its own portion of M_{shared}. This memory design relies on the fact that simply because a thread has access to a piece of memory does not mean that it will read from that memory. The threads in $\mathbf{T}$ are designed so that they only read the data they need to calculate their values of $\mathbf{x}_{n+1}$ from the shared memory. This design prevents unnecessary cache line reads, and therefore keeps memory accesses to a minimum.

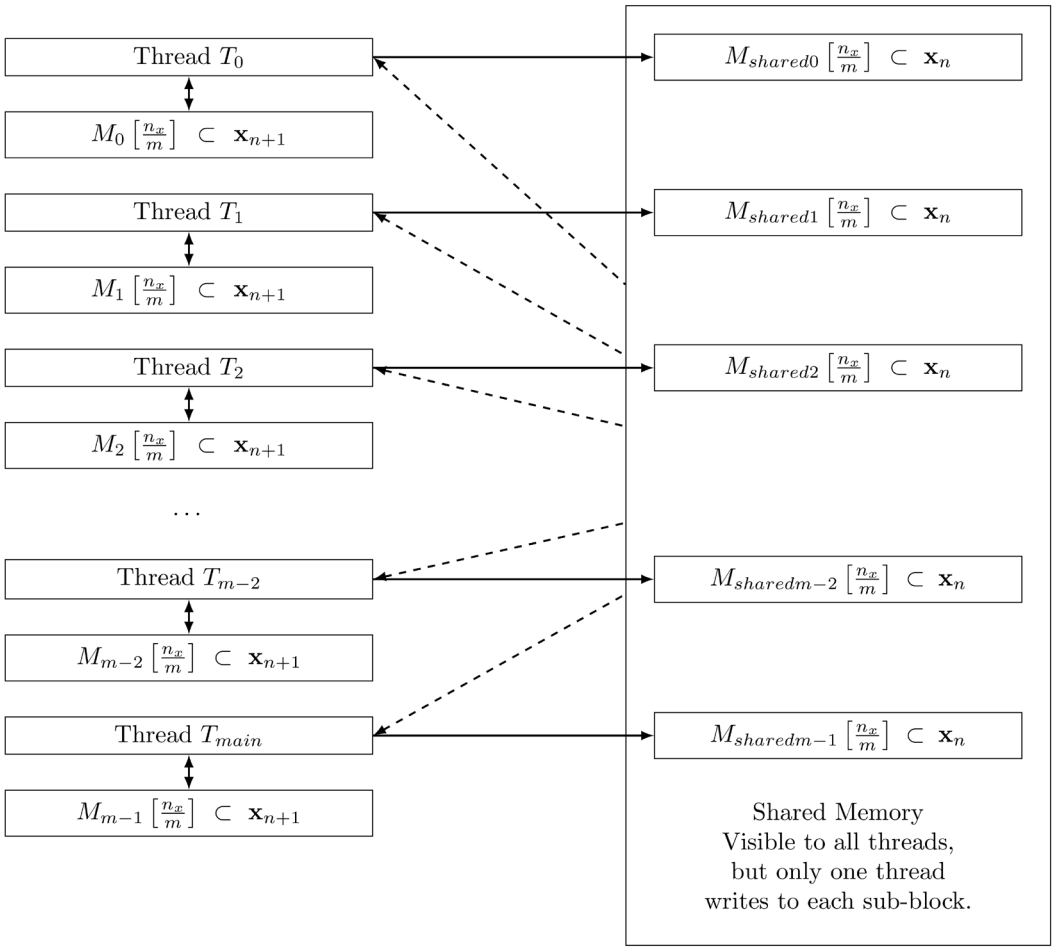

Figure 17. Figure describing Full Partitioned Memory Simple Agglomeration memory structure using 1 thread per CPU core. The dashed lines indicate a read-only relationship.

5.3.3. Experimental Runs to Evaluate Speedup

The relative speedups compared to the serial case are shown in **Table 1** and **Table 2**. These tables do not include the standard deviations because all of the standard deviations are on the order of 10^{-2}. **Figure 18** presents the relative speedups across all models.

From **Table 1** we can see that the model with 288 state variables did not produce a speedup compared to the serial case. Since the other models did produce a speedup, the lack of speedup on the smaller model is likely due to the model not having enough computational work to overcome the overhead associated with parallelization.

Table 2 shows the relative speedup when simulating the complex RLC model with 804 state variables (**Figure 9**). For most thread values Minimum Sharing matches the serial simulation, and in a few cases, such as when the model has fast time constants and is simulated using 4 threads, is able to surpass the serial simulation performance and provide a speedup of up to 1.44. Simple Agglomeration provided a small speedup when using the model with fast time constants, but was only able to match the serial simulation when simulating the model with slow time constants.

We note that Minimum Sharing and Simple Agglomeration have very different wall clock times when they are run only using one thread. This is an unexpected result because the memory differences between the two partitioning approaches should not come into play when only using one thread. The likely reason for the difference in single threaded simulation time is differences in implementation. Version 1 uses C-style structs for sharing data, and version 2 uses C-style arrays. Indexing into an array requires pointer arithmetic, which takes extra time, that is not present when using structs.

We also note that both of the Full Partitioned Approaches perform better on the complex RLC model with

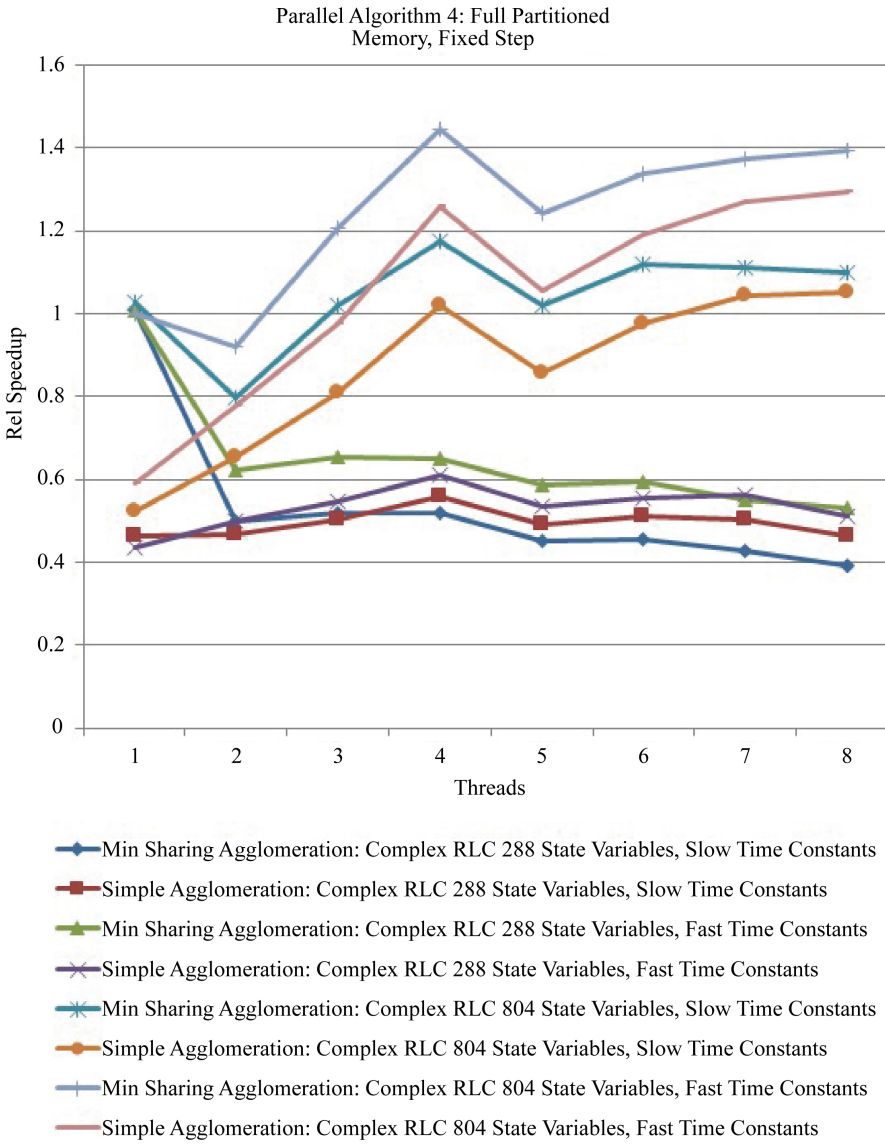

Figure 18. Figure showing the relative speedups across the models for the Full Partitioned Memory algorithms.

Table 1. Relative speedups for Full Partitioned Memory approaches versions 1 and 2 on the complex RLC model with 288 state variables.

Total Threads	Slow Time Constants		Fast Time Constants	
	Min. Sharing	Simple Agglom.	Min. Sharing	Simple Agglom.
1	1.01	0.46	1.01	0.43
2	0.50	0.47	0.62	0.50
3	0.52	0.50	0.66	0.55
4	0.52	0.56	0.65	0.61
5	0.45	0.49	0.58	0.53
6	0.45	0.51	0.59	0.56
7	0.43	0.50	0.55	0.56
8	0.39	0.46	0.53	0.51

Table 2. Relative speedup and standard deviations for Full Partitioned Memory approaches versions 1 and 2 on the complex RLC model with 804 state variables.

Total Threads	Slow Time Constants		Fast Time Constants	
	Min. Sharing	Simple Agglom.	Min. Sharing	Simple Agglom.
1	1.03	0.52	1.00	0.59
2	0.80	0.65	0.92	0.78
3	1.02	0.81	1.21	0.97
4	1.17	1.02	1.44	1.26
5	1.02	0.86	1.24	1.05
6	1.12	0.98	1.34	1.19
7	1.11	1.04	1.37	1.27
8	1.10	1.05	1.39	1.29

804 state variables using fast time constants than slow time constants. A possible explanation is due to the equations in $\mathbf{f}_{IFE}$, for the fast parameters, are more complicated than the equations for the slow parameters. This extra complexity, essentially just the presence of parameter values scaling the state variable values, means that the processor has more computational work to solve each equation and therefore the ratio of cache line reads to computational work goes down, and the cache line reads have less of an opportunity to dominate the run time. This analysis seems tenuous, but since these methods use fixed step integration, the change in model behavior has no effect on the integration (because the step size does not change as a result of system dynamics), and the only real difference between the fast and slow parameter values is the presence of the parameter value terms in the integration equations.

In **Figure 18**, and in the above tables, we also see that the speedup drops between threads 4 and 5. This is due to the hardware multithreading built into the processor. At 5 threads one of the CPU cores needs to run two threads instead of just one, and, because the cache is allocated to a CPU core, not to a thread, the threads must fight for cache resources. It is also possible that the OS moves the extra thread from core to core to try and balance the workload assigned to each core. But this ends up hurting performance because each time the thread is moved it must populate the cache on the new core, and is, therefore, not able to realize the benefits of using a cache.

We also see in **Figure 18** that in some instances there is a performance drop between using one thread and two threads. We are unsure what causes this.

We performed a mean-squared error analysis on our simulation results for both Minimum Sharing and Simple Agglomeration. The MSE calculations compared the time trajectory data of a single threaded fixed step simulation and a parallel simulation using 8 threads. In all cases where the simulation step sizes were small enough to produce a stable simulation, the error was on the order of 10^{-16} or smaller, so we did not include the mean square error results here.

5.4. Parallel Algorithms Type 5: Full Partitioned Memory, Variable Step Simulation

A variable step simulation is likely to provide better simulation performance than a fixed-step simulation. This algorithm tests a parallel version of a variable step solver, using a full partitioned memory approach, to determine if further speedups can be found.

We use a Runge-Kutta-Fehlberg4,5 solver from the GNU Scientific Library [23] as our variable step solver. It is implemented using a traditional simulation method, as shown in **Figure 2** and described in Section 2.2. To use the solver the user provides functions to calculate the system state variable derivatives, $\dot{\mathbf{x}}_n$, and the system Jacobian matrix. The user also provides the solver with an output interval detailing how frequently the user wants to receive updates on the state variable derivatives. Controlling the simulation time step is left up to the

solver (see Section 4.1.2).

To create a partitioned variable step solver we divide the model into ℓ independent pieces and assign a solver to each independent piece of the model. Each solver sets its own step size, and synchronization of the state variables occurs at pre-defined output points. To partition a model into ℓ independent pieces we modify the model partitioning in Equation (16) to:

$$\begin{aligned}
\dot{\mathbf{x}}_{n_0} &= \mathbf{f}_0\left(t_n, \mathbf{x}_{n_0}, \mathbf{w}_n, \mathbf{x}_{n_1_prev}, \cdots, \mathbf{x}_{n_(\ell-1)_prev}\right) \\
\dot{\mathbf{x}}_{n_1} &= \mathbf{f}_1\left(t_n, \mathbf{x}_{n_1}, \mathbf{w}_n, \mathbf{x}_{n_0_prev}, \mathbf{x}_{n_2_prev}, \cdots, \mathbf{x}_{n_(\ell-1)_prev}\right) \\
&\vdots \\
\dot{\mathbf{x}}_{n_\ell-1} &= \mathbf{f}_{\ell-1}\left(t_n, \mathbf{x}_{n_(\ell-1)}, \mathbf{w}_n, \mathbf{x}_{n_0_prev}, \cdots, \mathbf{x}_{n_(\ell-2)_prev}\right),
\end{aligned} \tag{27}$$

where the $\mathbf{x}_{n\_x\_prev}$ values are the state variable values calculated by the other solvers at the previous synchronization point. The primary difference between Equation (16) and the above Equation (27), is in that in Equation (27) the functions of $\mathbf{f}$ are required to use values of the state variables, $\mathbf{x}_n$, at the last synchronization point to calculate the state variable derivative values, $\dot{\mathbf{x}}_n$. In Equation (16) the functions $\mathbf{f}$ use the current values of the state variables to calculate the state variable derivative values. This partitioned system, where each RKF4,5 solver independently sets its own time step values, is a form of Multi-Rate Integration, which is investigated in [31] and [32].

The $\mathbf{x}_{n\_x\_prev}$ values are stored in a shared memory that is readable by all threads, but only one thread will write to a block of *prev* values to avoid cache line sharing problems. The synchronization points will update the *prev* values according to:

$$\begin{aligned}
\mathbf{x}_{n_0_prev} &= \mathbf{x}_{n_0} \\
\mathbf{x}_{n_1_prev} &= \mathbf{x}_{n_1} \\
&\vdots \\
\mathbf{x}_{n_(\ell-1)_prev} &= \mathbf{x}_{n_(\ell-1)}.
\end{aligned} \tag{28}$$

Each independent RKF4,5 solver is responsible for integrating its portion of state variable derivative values:

$$\begin{aligned}
RKF4,5_0\left(\dot{\mathbf{x}}_{n_0}\right) &\rightarrow \mathbf{x}_{n_0} \\
RKF4,5_1\left(\dot{\mathbf{x}}_{n_1}\right) &\rightarrow \mathbf{x}_{n_1} \\
&\vdots \\
RKF4,5_{\ell-1}\left(\dot{\mathbf{x}}_{n_(\ell-1)}\right) &\rightarrow \mathbf{x}_{n_(\ell-1)}.
\end{aligned} \tag{29}$$

Due to the fact that the solvers are independent, the solvers will be forced to integrate their set of $\dot{\mathbf{x}}_n$ values without a full view of the system state. This can lead to errors in the integration and incorrect simulation results because the solvers are designed to integrate an entire system, not simply a small part of it (however, we did not have a problem with accuracy in our simulations). Another source of errors is that the there will need to be an individual Jacobian function calculated for each solver based on the subset of $\mathbf{f}$ for which the solver is responsible. There will be state variables used in the equations of the subset of $\mathbf{f}$ that are not a part of a solver's assigned subset of $\mathbf{x}_n$. Therefore, when the Jacobian function is calculated, these extra variables will be treated as constants instead of as system state variables. This will make the Jacobian matrix incomplete because it will not include all of the information that it would if the system were not partitioned. As an example of this, consider this system of state equations:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \\ a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}. \tag{30}$$

Since this system is so simple, the Jacobian matrix of this system matches the system matrix above:

$$J = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \\ a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix}. \tag{31}$$

However, if this system is divided into two independent systems, as we do for the partitioned RKF4,5 parallel approach, the equations for the first system become:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 & 0 \\ 0 & b_2 & c_2 & d_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_{3\_prev} \\ x_{4\_prev} \end{bmatrix} \tag{32}$$

$$J = \begin{bmatrix} a_1 & b_1 \\ 0 & b_2 \end{bmatrix}, \tag{33}$$

and the equations for the second system become:

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} a_3 & 0 & c_3 & d_3 \\ a_4 & b_4 & 0 & d_4 \end{bmatrix} \begin{bmatrix} x_{1\_prev} \\ x_{2\_prev} \\ x_3 \\ x_4 \end{bmatrix} \tag{34}$$

$$F = \begin{bmatrix} c_3 & d_3 \\ 0 & d_4 \end{bmatrix}. \tag{35}$$

where the $\_prev$ values are the state variable values from the previous synchronization point. When comparing the full system description in Equation (30) to the two partial descriptions in Equations (32) and (34) we can see that the full description is able to use current values of the state variables to calculate the state variable derivatives, while the partitioned models have to use outdated state variable values to calculate their state variable derivatives. Also notice that the two Jacobian matrices for the partitioned systems, Equations (33) and (35), are much smaller and do not contain all of the data present in the full matrix, Equation (31). Specifically, the Jacobian matrix in Equation (33) is missing the partial derivatives with respect to x_3 and x_4, and the Jacobian matrix in Equation (35) is missing the partial derivatives with respect to x_1 and x_2.

Taken together, the two factors of out of date data and an incomplete Jacobian matrix can lead to significant problems for this simulation approach. The only option we have to control this potential problem is to reduce the synchronization time interval, because a faster synchronization time, like a smaller step size, will help to reduce errors in simulation. Producing an accurate simulation data will require balancing between setting the synchronization interval small enough that the errors in the approximations are small, but not setting the synchronization interval so small that there is no performance benefit.

The memory structure of the simulation is shown in **Figure 19**. The structure is similar to the full partitioned memory version 2, but there are two local memory blocks for each thread: one for $\mathbf{x}_n$ and one for $\dot{\mathbf{x}}_n$. The threads read and write to their local memory blocks at every function evaluation. The threads only update the shared memory block at the synchronization points, but can read from it at any time.

The program flow for the partitioned RKF4,5 simulation is somewhat more complex than the previous program flows and is shown in **Figure 20**. The synchronization points, where we update the shared memory with the most recent values of $\mathbf{x}_n$, only happen when the RKF4,5 solvers pause to provide output. We do not synchronize after every time step like we do for the fixed step algorithms detailed above. In this program flow, like the full partitioned memory approaches described above, all of the threads in $\mathbf{T}$ are responsible for integrating a portion of $\dot{\mathbf{x}}_n$, and they all synchronize their assigned variables.

Experimental Runs to Evaluate Speedup

This approach generally performed very well. The relative speedups for the models in **Figure 8** and **Figure 9** are shown in **Table 3**. **Figure 21** presents the relative speedups across all models. The relative speedup numbers

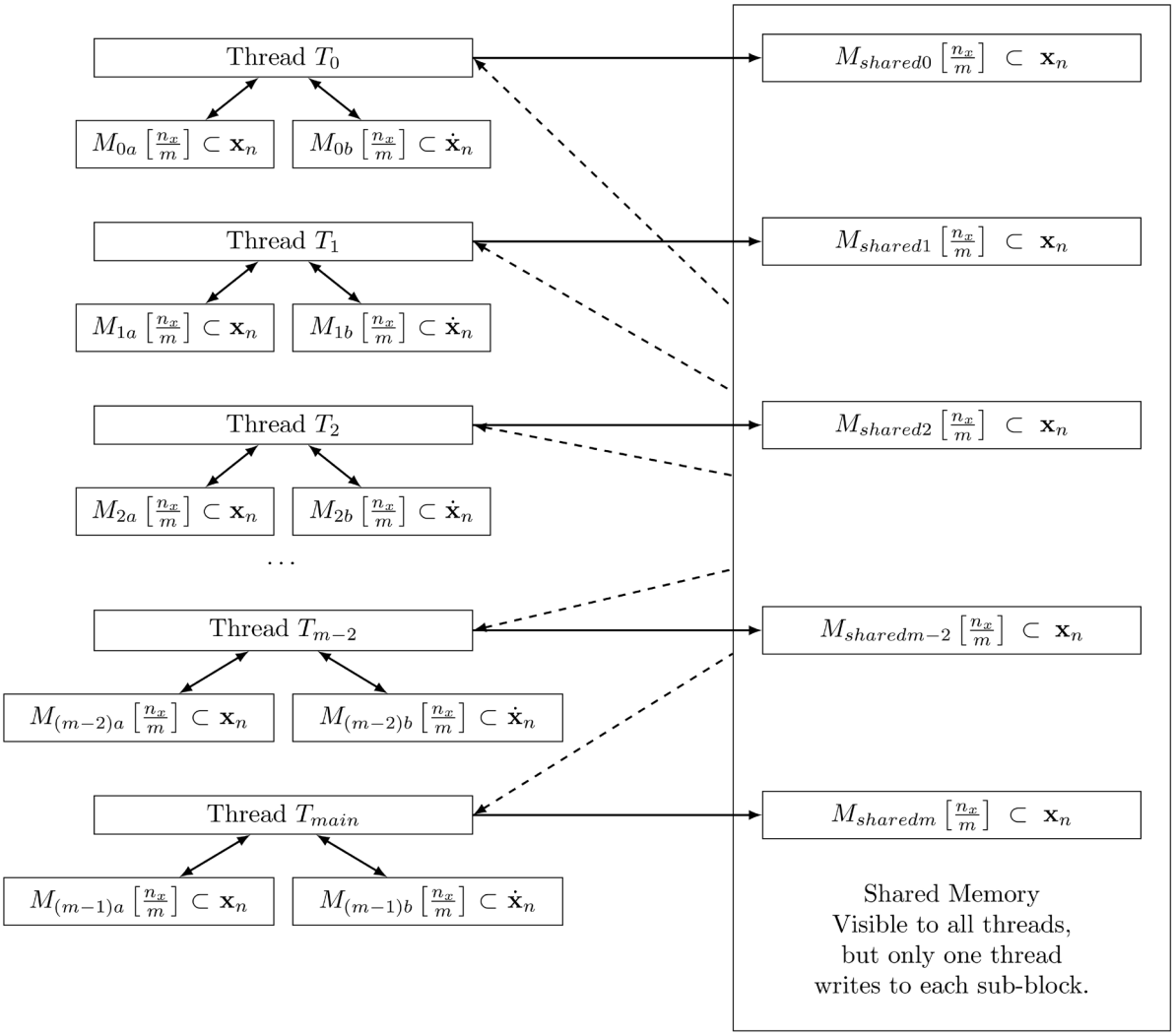

Figure 19. Figure describing full partitioned memory implementation of a partitioned RKF4,5 simulation when using agglomeration and 1 thread/RKF4,5 solver per CPU core. The dashed lines indicate a read-only relationship.

Table 3. Relative speedups and standard deviations for the partitioned RKF4,5 approach on the complex RLC models with 288 and 804 state variables, for models with both slow and fast time constants.

Total Threads	RLC with 288 State Variables		RLC with 804 State Variables	
	Slow TC	Fast TC	Slow TC	Fast TC
1	1.00	1.00	1.00	1.00
2	1.75	1.76	1.81	1.93
3	2.17	2.30	2.43	2.72
4	2.81	2.84	3.14	3.56
5	2.41	2.17	2.46	2.12
6	2.68	2.41	2.87	2.01
7	2.60	2.64	1.44	0.32
8	2.53	2.81	1.51	0.33

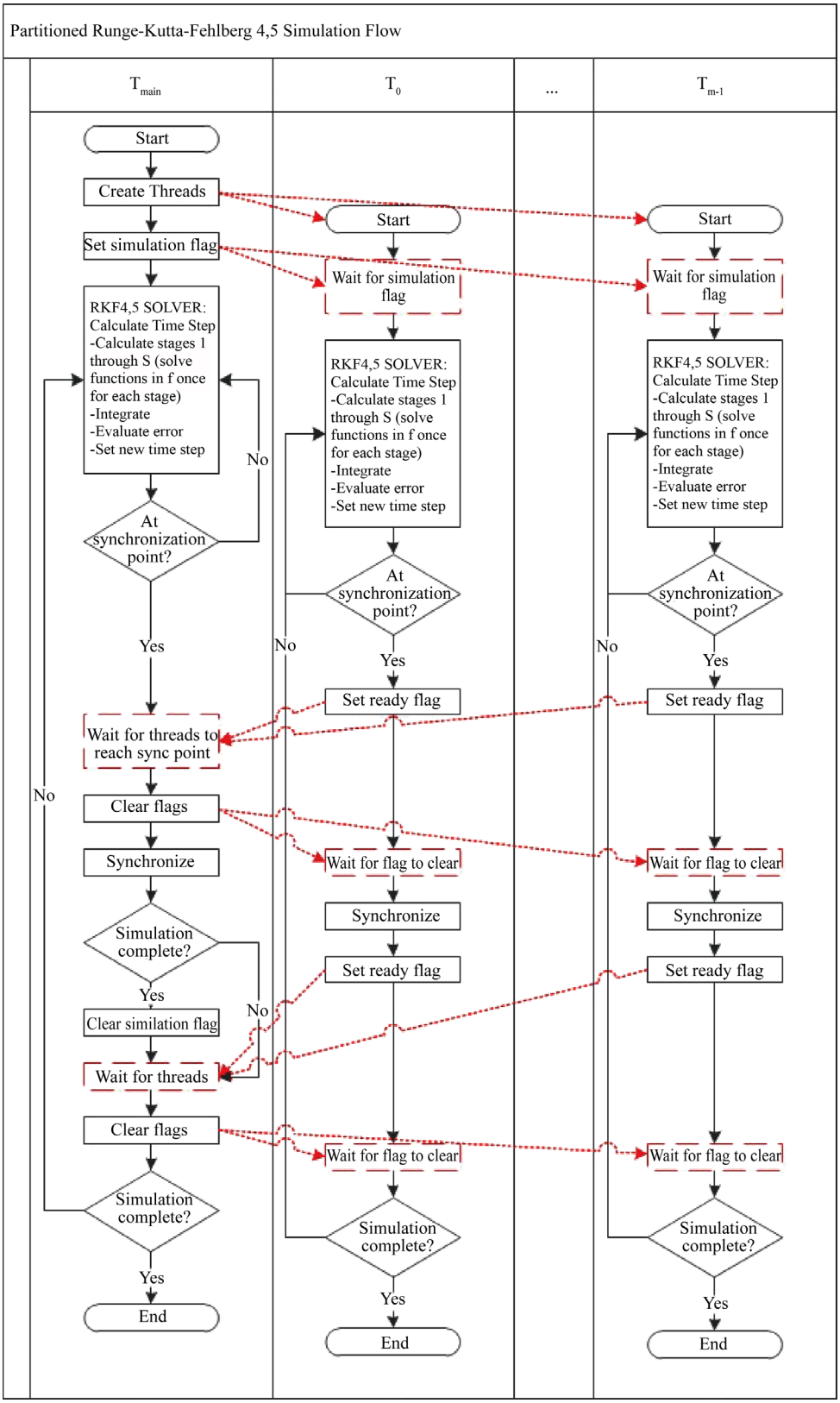

Figure 20. Partitioned RKF4,5 parallel simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

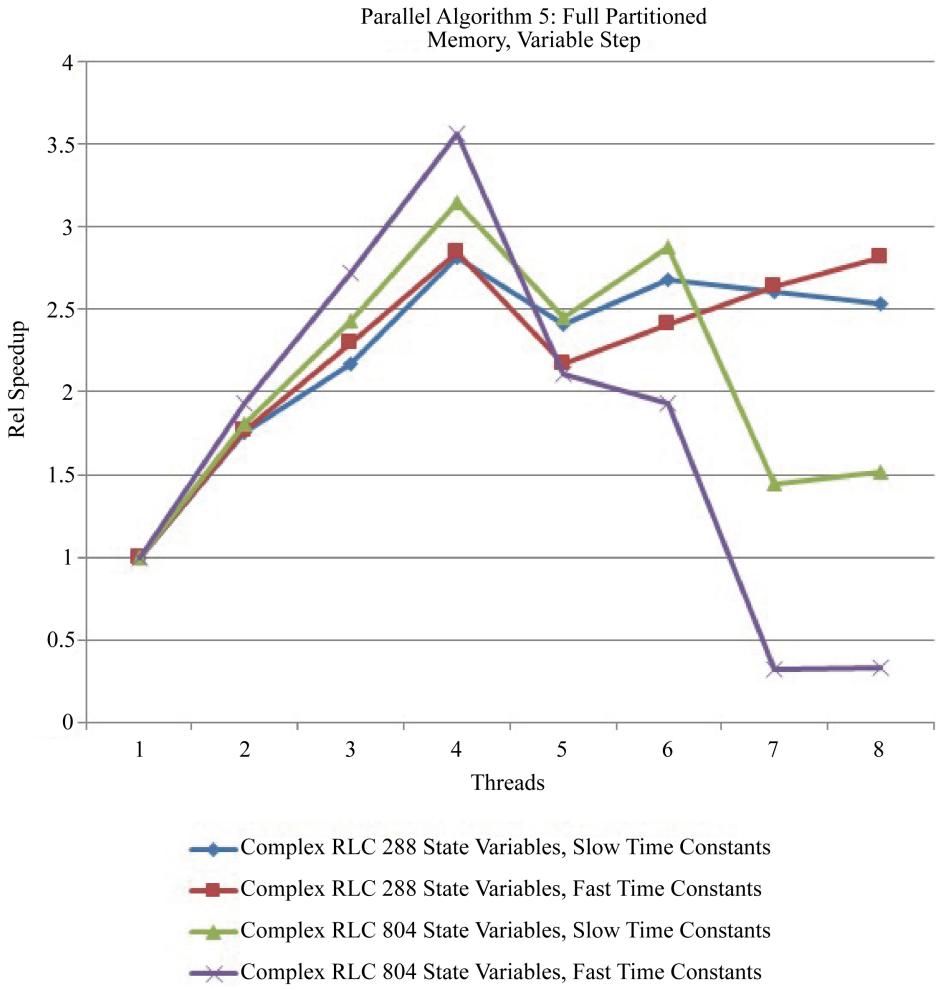

Figure 21. Figure showing the relative speedups across the models for the variable step Full Partitioned Memory algorithms.

were calculated by comparing the simulation times to a single threaded, serial, variable step simulation.

These results show that for the complex RLC model with 288 state variables the partitioned RKF4,5 method was able to match the serial simulation or provide a speedup when using 2 through 8 threads. The models with fast and slow time constants had similar performance, with the best speedup of approximately 2.8 coming when 4 threads were used.

For the complex RLC model with 804 state variables and slow time constants the partitioned RKF4,5 method was able to provide a speedup when using 2 through 8 threads. When simulating the model with fast time constants the algorithm was able to provide a speedup when using 2 through 6 threads. When using 7 and 8 threads, the model with fast time constants did not provide a speedup.

We performed a Mean Squared Error (MSE) analysis for the RLC models, by comparing the simulation time trajectory data between a serial fixed step simulation and a parallel variable step simulation. When we calculated the MSE for a simulation using our chosen synchronization interval, the mean square error was very small, with the largest error on the order of 10^{-5} and most errors much smaller than that. Since the errors were so small, they are not included in this paper. Also, the small error values indicate that the process of parallelizing a simulation does not negatively affect the accuracy of the simulation. This shows that our concerns about breaking a model into independent partitions were not justified for this set of models.

5.5. Parallel Algorithms Type 6: Algebraic Loop Simulation

Algebraic loops are very common in modern complex models, algebraic loops are typically solved using the

Newton Iteration method [11]. For non-linear algebraic loops the number of iterations is unknown, because it depends on the convergence time of the Newton Iteration, which is dependent on the initial guess provided to the algorithm. It is reported that nonlinear loops have a computational complexity of $O(n^3)$ [33] [34], so our approach to parallelize the simulation of a model that contains algebraic loops will focus on solving the algebraic loops in parallel, and then perform the numerical integration serially. We use KINSOL from the SUNDIALS solver library [35] from Lawrence Livermore National Laboratory [36] as our algebraic loop solver. We use the same fixed step (Forward Euler, Equation (3)) and variable step (RKF4,5) numerical integration methods that we used in the above tests for experiments on models with algebraic loops.

The models have a number of properties that allow us to simplify the simulation algorithms. The first property is that the variables calculated by the loops are from the set $\mathbf{w}_n$ from Equation (1). All of the loops need to be solved before the system state variables are calculated, because the equations in $\mathbf{f}_x$ may depend on the values of $\mathbf{w}_n$ calculated in the loops in order to calculate the state variable derivatives, $\dot{\mathbf{x}}_n$. Another property is that all of the loops are independent of each other, and therefore can be solved in any order for each time step. In addition, each thread is assigned approximately the same number of loops to solve for each time step. Once all the loops are solved, the state variable derivatives are calculated and then integrated to end the simulation time step. The program flow is shown in **Figure 22**. We applied this method to the fixed step and variable step simulation algorithms for our experiments.

The memory management scheme employed is shown in **Figure 23**. Each thread is assigned a group of algebraic loops to solve, $0, \cdots, pT$, where pT is the number of loops assigned to a thread. There are m groups of loops created, where m is the number of threads being used. Each loop has its own dedicated memory block that has the same number of variables as there are equations in the loop (the size of the loop-specific memory blocks was omitted in the figure to save space). Solving the loops will require data from the state variables, $\mathbf{x}_n$, so each thread needs read access to the main shared memory so it can pass that data to the loop solvers. The main thread is assigned a subset of algebraic loops to solve, and is also responsible for integrating the state variable derivatives (see **Figure 22**). Some of the calculations for the state variable derivatives may depend on the variables calculated from the algebraic loops, which means that the main thread needs read access to each loop specific memory block.

When developing this approach for parallelizing the solving of algebraic loops, we elected to not parallelize the variable step solver, because parallelizing it would likely add much more computational work to the simulation, and likely end up slowing down the simulation instead of making it faster. The reason for the potential slowdown is due to the fact that solving the algebraic loops is embedded the function $\mathbf{f}_x$ from Equation (1). When partitioning the model as described in Section 5.4 in Equation (27), some of the loops in $\mathbf{f}_x$ will be duplicated and assigned to multiple partitions because of dependencies between the loops and the functions to calculate the state variable derivatives, $\dot{\mathbf{x}}_n$. As an example of the possible extra work potentially involved in parallelizing the integration of the RKF4,5 method, consider the computational work to integrate across a timestep for a system with 4 state variables. The RKF4,5 solver is a 6-stage method [37], meaning there are 6 function evaluations to integrate across a time step. When integrating a model that has 4 algebraic loops, to integrate the model across one time step will require solving the 4 loops 6 times, for a total of 24 loop evaluations. If we partition the system into 4 components, with one state variable in each component, as described in Section 5.4, in the worst case integrating each partition will require solving each of the 4 loops. Therefore, integrating the entire model across 1 time step (even though the time step sizes may be different for each partition) will require $4 \text{ loops} \times 4 \text{ partitions} \times 6 \text{ stages} = 96$ loop evaluations. Since loops are so computationally expensive to solve, adding the extra loop evaluations above the serial case will likely increase the amount of time required to solve the system. This is only a worst case analysis, however, and a real model will not likely suffer from this problem. We leave partitioning the integration of a system with algebraic loops for future work.

Experimental Runs to Evaluate Speedup

The relative speedups and standard deviations of the models with algebraic loops are shown in **Table 4** and **Table 5**. **Figure 24** and **Figure 25** present the speedups graphically. To calculate these speedups, the fixed step parallel simulations were compared to a fixed step serial simulation, and the variable step parallel simulations were compared to a variable step serial simulation. Solving the loops in parallel generally had very good per-

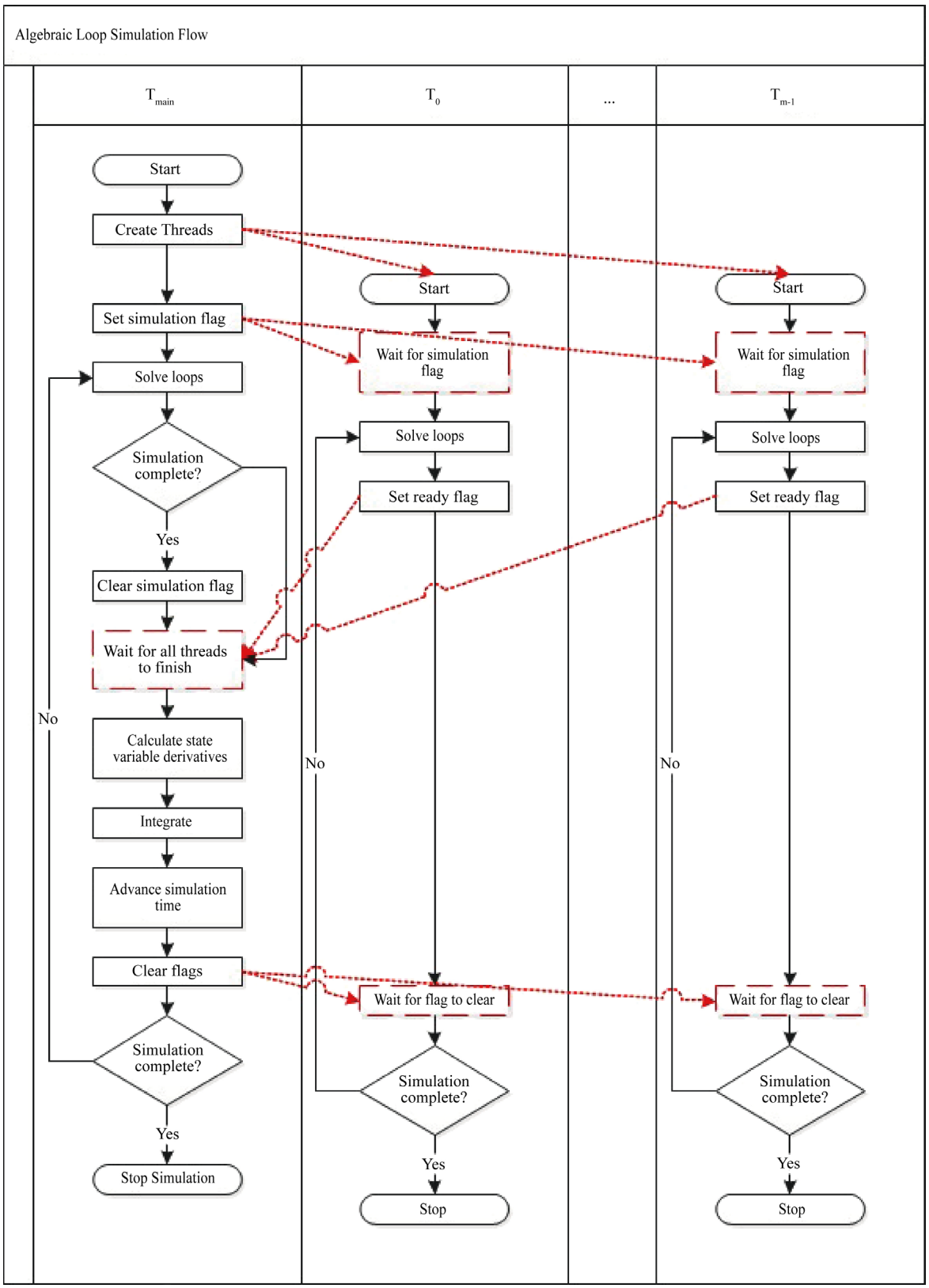

Figure 22. Algebraic loop simulation program flow. The red dashed lines indicate communication between threads, and the boxes with red dashes are wait states.

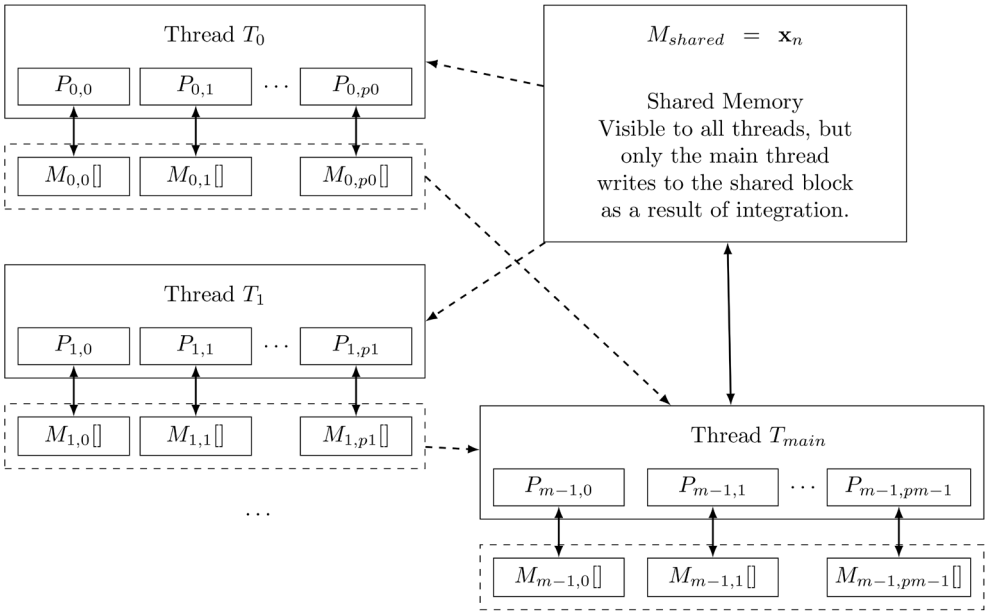

Figure 23. Figure describing memory implementation parallel loop simulation. The dashed lines indicate a read-only relationship.

Table 4. Relative speedup and standard deviation for Parallel Algebraic Loop approach using fixed step and variable step integration methods on the complex RLC model with loops and 288 state variables.

Total Threads	Slow Time Constants		Fast Time Constants	
	Fixed Step	Variable Step	Fixed Step	Variable Step
1	1.00	1.00	1.00	1.00
2	1.59	1.81	1.85	1.76
3	2.06	2.33	2.50	2.36
4	2.32	3.06	3.32	3.20
5	1.99	2.63	2.69	2.49
6	1.99	2.43	2.95	2.62
7	1.70	2.46	3.32	3.04
8	1.67	2.40	3.31	2.97

Table 5. Relative speedup and standard deviation for Parallel Algebraic Loop approach using fixed step and variable step integration methods on the complex RLC model with loops and 804 state variables.

Total Threads	Slow Time Constants		Fast Time Constants	
	Fixed Step	Variable Step	Fixed Step	Variable Step
1	1.00	1.00	1.00	1.00
2	1.46	1.72	1.92	1.88
3	1.95	2.42	2.52	2.11
4	2.38	3.14	3.33	2.88
5	1.83	2.56	2.72	2.42
6	2.14	2.93	2.94	2.53
7	2.10	2.91	3.46	2.67
8	2.11	3.11	3.49	2.43

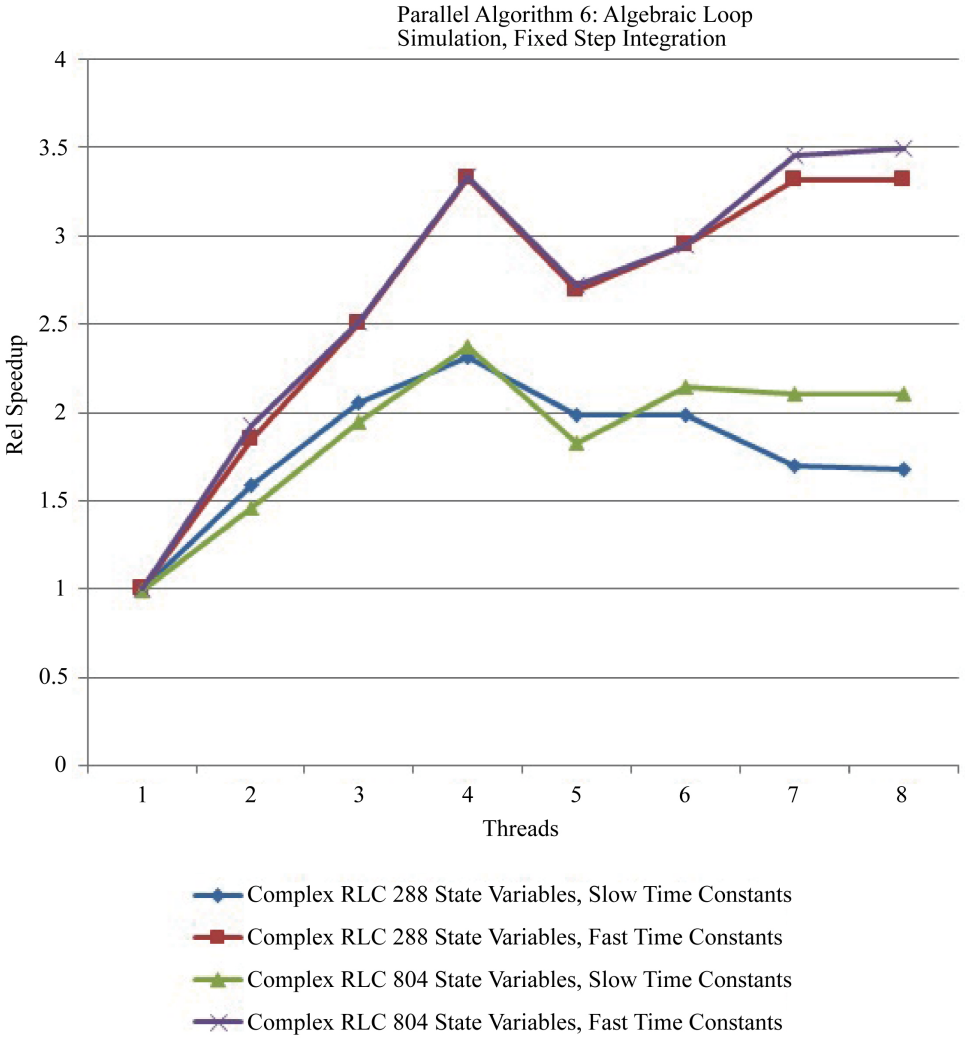

Figure 24. Figure showing the relative speedups across the models with algebraic loops using fixed step numerical integration.

formance, and every test, except for the single threaded experiments which served as a control, showed a speedup.

The variable step simulation performed slightly better than the fixed step simulation on the models with slow time constants. On these models the variable step simulation achieved a maximum speedup of slightly more than 3 when using 4 threads, while the fixed step simulation only achieved a speedup of about 2.3 when using 4 threads.

The fixed step and variable step simulations had near identical performance on the complex RLC model with 288 state variables and fast time constants; when using 4 threads the fixed step simulation achieving a maximum speedup of 3.3 and the variable step simulation achieving a maximum speedup of 3.2. The results between the fixed step and variable step simulations are likely similar because the processing of the algebraic loops is dominating the simulation run time, and therefore there is no advantage to using a fixed step or variable step solver.

The complex RLC model with 804 state variables and fast time constants provided interesting results because the fixed step simulation performed better than the variable step simulation. The fixed step simulation achieved a speedup of 3.5 when using 8 threads (it achieved a speedup of 3.33 when using 4 threads), while the variable step simulation achieved a maximum speedup of only 2.88 when using 4 threads. This is likely due to, again, the processing of the algebraic loops dominating the computation time, which allows the computational efficiency of the fixed step simulation to provide a speedup over the variable step simulation.

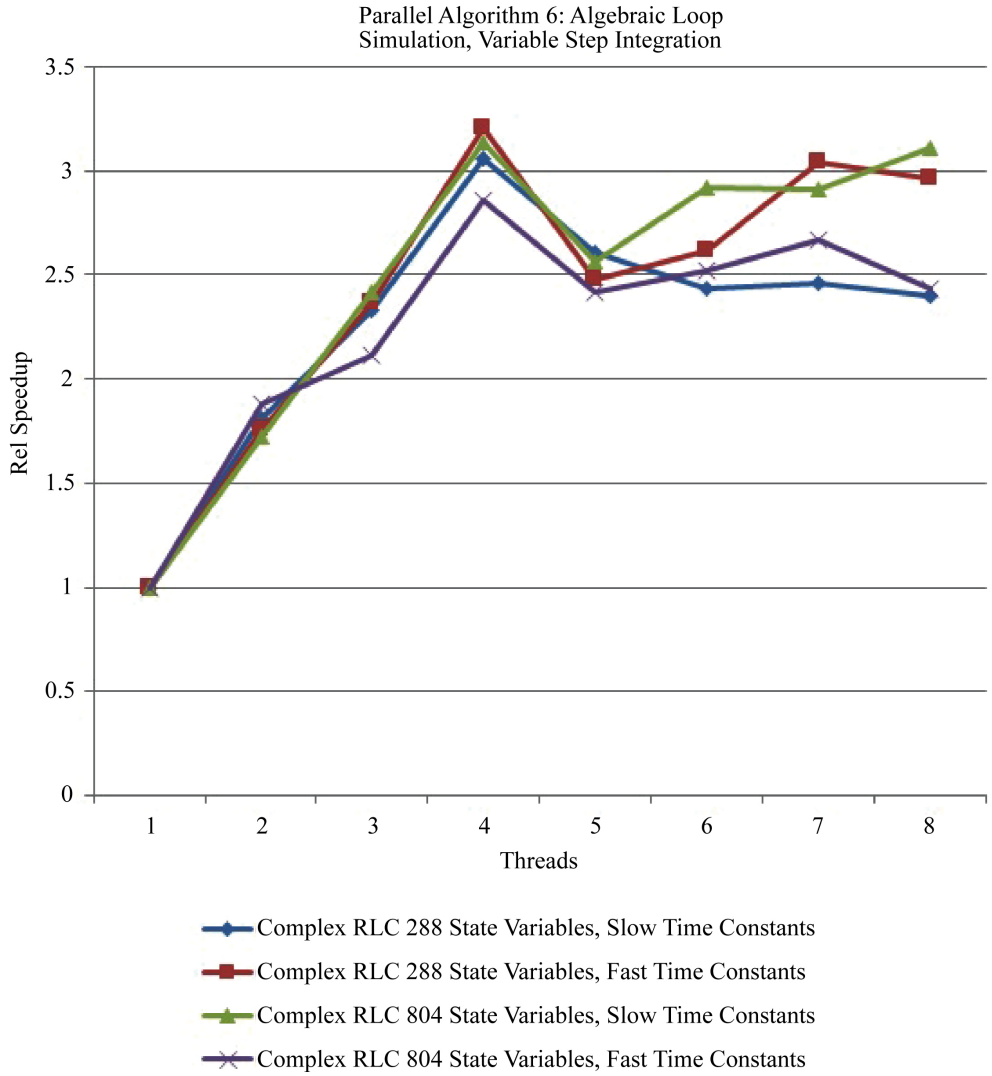

Figure 25. Figure showing the relative speedups across the models with algebraic loops using variable step numerical integration.

6. Results Summary and Discussion

This section summarizes our parallel algorithms, and experiment results, and discusses the conclusions we are able to draw from those results. As a review, **Table 6** presents the key differences between the different parallel simulation algorithms.

Results Review

Table 7 presents a summary of the best relative speedups produced by each parallel algorithm. Our algorithms show a steady improvement, except for parallel algorithm type 3, from **Algorithm 1-5**. In **Algorithm 1** our best speedup was on the order of 10^{-3}, which means our parallel implementation was significantly slower than the serial case. In this algorithm we were creating more threads than the CPU and the operating system were able to efficiently handle and most of the processing time of the simulation was spent on the overhead of switching between the threads instead of on advancing the simulation.

In **Algorithm 2** we reduced the number of threads so that the maximum number of threads used for the simulation was equal to the number of CPU cores available on our processor. This method did not provide a speedup, but it was able to match the serial simulation time. The lack of speedup for **Algorithm 2** was caused by

Table 6. Summary of parallel algorithms.

Algorithm	Memory Structure	$\lvert\mathbf{T}_{spawn}\rvert$	Role of $\mathbf{T}_{spawn}$	Role of T_{main}	Agglomeration
Type 1	Full Shared	n_x	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	None
Type 2	Full Shared	$m-1$	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	Simple and Smart
Type 3	Partial Partitioned	$m-1$	Calculate $\mathbf{f}_{IFE}$	Merge $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$	Simple
Type 4	Full Partitioned	$m-1$	Calculate $\mathbf{f}_{IFE}$ and Merge	Calculate $\mathbf{f}_{IFE}$ and Merge	Simple and Minimum Sharing
Type 5	Full Partitioned	$m-1$	Calculate $\dot{\mathbf{x}}_n$ and Merge	Calculate $\dot{\mathbf{x}}_n$ and Merge	Simple
Type 6	Full Partitioned	$m-1$	Solve algebraic loops	Solve algebraic loops and integrate	Simple

Table 7. Summary of results.

Algorithm	Best Performance		
	Rel Speedup	Threads	Model
Type 1	2.0×10^{-3}	8	RLC 288 State Variables, Slow Time Constants
Type 2	0.61	8	RLC 804 State Variables, Fast Time Constants
Type 3	0.41	4	RLC 804 State Variables, Fast Time Constants
Type 4	1.44	4	RLC 804 State Variables, Fast Time Constants
Type 5	3.56	4	RLC 804 State Variables, Fast Time Constants
Type 6	3.49	8	RLC 804 State Variables, Fast Time Constants

not considering the CPU cache in our experiments, and the different threads had to wait for an update to their cache lines instead of completing the simulation.

In **Algorithm 3** we attempted to avoid the problems of cache line sharing by creating a memory structure that would prevent cache conflicts between the threads. Unfortunately this algorithm performed worse than **Algorithm 2** due to our memory structure significantly increasing the amount of work T_{main} needed to complete each time step. This extra work by T_{main} prevented **Algorithm 3** from producing a speedup.

In **Algorithm 4** and **Algorithm 5** we further enhanced our memory structure so that the problems seen in **Algorithm 3** were solved, and we reduced the workload of T_{main} and expanded the role of the threads in $\mathbf{T}_{spawn}$ so that all of the work of the simulation was spread across all threads. **Algorithm 4** and **Algorithm 5** use the same memory structure and the same responsibilities of each thread, but **Algorithm 4** uses a fixed step numerical integration method, and **Algorithm 5** uses a variable step numerical integration algorithm. These two algorithms provided good speedups compared to the serial simulation case. The best speedup of **Algorithm 4** was 1.44, the best speedup of **Algorithm 5** was 3.56, and both on the RLC models with 804 state variables and fast time constants.

Algorithm 6 only parallelized the algebraic loops, it did not parallelize the simulation as a whole as the previous algorithms. However, in developing **Algorithm 6** we applied the lessons learned in **Algorithms 1-5**. These lessons include: limiting the number of threads created, avoiding cache line sharing, and evenly distributing the processing across all threads. **Algorithm 6** also provided a good speedup compared to a serial simulation, and the largest speedup was 3.49 on the complex RLC model with 804 state variables and fast time constants. Further conclusions are discussed in the next section.

7. Conclusions

This research provided a number of interesting and practical conclusions about parallel simulation. These will be detailed in the following sub-sections.

7.1. Conclusion 1: Model Size

The first conclusion that we are able to draw from these results is that there is a model size threshold below which it is difficult to draw a benefit from parallelization. We are parallelizing within a time step, so the amount of time taken per time step is the real barrier that we are trying to beat with parallelization. Our RLC model with 288 state variables when using parameters that created slow time constants had a time per time step of about 500 ns on our hardware; when using the parameters that created the fast time constants the time per time step is about 700 ns. On this model we only saw a speedup when using variable step integration; the best performance for the fixed step integration was 0.56 when using the slow parameters, and 0.61 when using the fast parameters. For the large RLC model the time per time step for the serial simulation is about 2 μs for the slow parameters and about 3 μs for the fast parameters, and the full partitioned memory method was able to produce a small speedup of about 1.2 for the slow parameters and of about 1.4 for the fast parameters. A better measurement is CPU clock cycles. On the CPU we used for our experiments, clocked to 3 GHz, 700 ns equates to approximately 2100 clock cycles on the CPU, while 3 μs is approximately 9000 clock cycles. Since the large model produced a speedup and the small model did not, we can determine that the minimum model size above which parallelization is practical is going to be just under 800 state variables, or about 9000 clock cycles for one time step, and the serial time of the time step needs to be on the order of microseconds.

7.2. Conclusion 2: Limit Threads to Number of Physical Cores on the CPU

In Section 3.1.2 we presented an experiment that shows that cache line sharing can have a significant impact on a program's run time (**Figure 5**). Another feature that can be drawn from that experiment is that hardware multithreading does not produce the same benefit as having separate physical CPU cores because the time to complete the experiment begins to increase when we start using 5 threads. We are using a 4-core CPU where each core is multithreaded and can support 2 threads. When using 4 cores the OS is able to partition one thread per CPU core. However, when using 5 threads one of the cores is going to have to run 2 threads simultaneously, which degrades performance. We see this in our experiments in Section 5, where there is a drop off in performance in moving from 4 threads to 5 threads. For most experiments the best performance was using 4 threads. The conclusion that we can draw from this data is that for simulation purposes hardware multithreading (see Definition 6), which is the technology used to support more than 1 thread on a single CPU core, produces worse performance than using the same number of threads as there are cores on the CPU, and it should be avoided. A possible reason for the lack of performance of hardware multithreading is due to both threads on the core having to share an L1 cache, and therefore the amount of cache available for each thread is reduced by half compared to the single threaded case. A second reason is that simulation is very processor intensive activity, and the hardware multithreading logic built into the CPU is not able to adequately partition the computational work of 2 threads onto the single processor built into the CPU core.

7.3. Conclusion 3: Memory Structure Should Support CPU Cache

Another conclusion that we can draw from Section 3.1.2 and from our experiment results in Section 5 is that minimizing cache line reads and avoiding cache line sharing is crucial to parallel program performance. These two factors prevented parallel simulation methods 1 and 2 from providing very good performance.

In the full shared memory approaches there was both cache line sharing between the threads of $\mathbf{T}_{spawn}$ and too many cache line reads for thread T_{main}. The problem with the cache line reads of T_{main} is that T_{main} was simply merging the variables of $\mathbf{x}_{n+1}$ into the variables of $\mathbf{x}_n$. It was not performing any calculation work on the variables, and therefore the time to perform the cache line reads dominated the time to perform the merge. The benefit of assigning each thread in $\mathbf{T}$ to perform its own merge step, as was done in the full partitioned methods, Sections 5.3 and 5.4, is that the variables to perform the merge are already in the cache for the thread and there is no need to read a large number of cache lines to perform the merge as required by the full shared memory and partial partitioned memory methods.

Another factor in the good performance of these methods is that the memory block that each thread in $\mathbf{T}$ wrote to did not share cache lines with any other block. We were able to guarantee this through the use of the align as keyword, introduced as a part of the C++11 language standard. Each memory block used the align as keyword to align the block of memory to a cache line boundary, by doing this no part of the memory block was on the same cache line as another block.

7.4. Conclusion 4: Division of Labor

Another important aspect to designing parallel simulation algorithms is to ensure that the computational work of a time step is divided evenly across the computational threads. This was a second major problem with our early parallel simulation methods. These methods assigned the computational work of solving and integrating the equations in $\mathbf{f}_{IFE}$ to the threads in $\mathbf{T}_{spawn}$ while the thread T_{main} was only responsible for merging $\mathbf{x}_{n+1}$ into $\mathbf{x}_n$ and for advancing the simulation time. Not only did this setup contribute to problems of too many cache line reads with too little computational work in T_{main} but it also meant that only thread T_{main} or threads $\mathbf{T}_{spawn}$ were active at any given time. There was a distinct back and forth flow between T_{main} and $\mathbf{T}_{spawn}$. This is not a good use of computational resources, as the time the threads spend waiting is time wasted. The full partitioned methods, Sections 5.3 and 5.4, and the parallel loop method, Section 5.5, solve this problem by including T_{main} in the computation of $\mathbf{f}_{IFE}$ and by including $\mathbf{T}_{spawn}$ in the merge process. By expanding the roles of both sets of threads we were able to avoid wasted time during the simulation.

7.5. Conclusion 5: Software Design

Simulations execute for a large amount of simulated time will likely require a large number of time steps. For our experiments, some models were run for 5 million time steps. At that number of iterations small inefficiencies in the simulation code, that would have been ignored or undetectable had they only been run once, can become a source of significant lost time. In a program such as a simulation where the same piece of code is repeated many times, every line of code in that program needs to be closely examined to make sure it is as efficient as possible.

8. Summary and Future Work

In this work, we presented a series of parallel simulation algorithms for ODEs that are specifically designed to accommodate the features of a modern multi-core CPU. Specifically, the algorithms consider the multiple CPU cores, the processor cache, and their interactions to derive maximum speedup. The final three of these algorithms, Full Partitioned Memory Fixed Step (Section 5.3), Full Partitioned Memory Variable Step (Section 5.4), and Algebraic Loop Simulation (Section 5.5), produced good speedup when compared to our serial test case. We also developed a series of experimental studies that allowed us to analyze the effectiveness of various multithreading and memory management schemes for parallel simulations. Since these algorithms are based on an ODE representation of the model behavior, they can be applied equally across models that cover multiple physical domains.

In the process of systematically developing these algorithms we derived a set of recommendations that apply to parallel simulation (Section 7) on modern multi-core processors. These conclusions include recommendations on the size of the model to be parallelized, the number of threads to use for the simulation, the memory management scheme to use, how to divide the computational work between the threads, and software optimizations to implement.

An important limitation of this work is that the models to be simulated must be reduced to ODE form. This results in a loss of the time trajectory data of the algebraic variables in the system, but the trajectory data for the state variables is maintained. The time trajectory data of these algebraic variables are typically preserved in traditional Modelica simulation [26]; however, by reducing the model to ODE form we save computation time during the simulation.

Another limitation is that our parallel algorithms are specifically designed for the parallel architecture of a multicore CPU. Applying the algorithms, unaltered, to a different parallel architecture, such as a SIMD architecture or a GPU, the algorithms will not perform as well.

A third limitation is that we only address a small subset of the Modelica language; more complicated Modelica models that include discrete mode transitions and conditional behavior, can force the simulation software to re-derive the system ODE equations during the simulation. Our algorithms depend on a fixed set of ODE equations, and do not allow those equations to change during run time. Therefore, we cannot support models that have those features without significant changes to our simulation architecture.

For future work, we would like to develop methods for formally solving the optimization problem of the model equations, expand our parallelization algorithms to support an additional numerical integration method,

apply our algorithms to a General Purpose GPU, parallelize the integration of our algebraic loop simulations, and apply intelligent load balancing when assigning algebraic loops to threads.

Acknowledgements

Initial research supported under DARPA META contract FA8650-10-C-7082. This support is greatly appreciated. The authors also wish to thank Zsolt Lattmann at Vanderbilt University for his help creating simulation models.

References

[1] Krogh, B.H., *et al.* (2008) Cyber-Physical Systems Executive Summary. CPS Steering Group, Arlington.

[2] Sztipanovits, J., *et al.* (2012) Toward a Science of Cyber-Physical System Integration. *Proceedings of the IEEE*, **100**, 29-44. http://dx.doi.org/10.1109/JPROC.2011.2161529

[3] Lee, E.A. (2008) Cyber Physical Systems: Design Challenges. 11*th IEEE International Symposium on Object Oriented Real-Time Distributed Computing* (*ISORC*), Orlando, 5-7 May 2008, 363-369. http://dx.doi.org/10.1109/isorc.2008.25

[4] Karsai, G. and Sztipanovits, J. (2008) Model-Integrated Development of Cyber-Physical Systems. *Software Technologies for Embedded and Ubiquitous Systems*, **5287**, 46-54. http://dx.doi.org/10.1007/978-3-540-87785-1_5

[5] Sangiovanni-Vincentelli, A. (2007) Quo Vadis, SLD? Reasoning about the Trends and Challenges of System Level Design. *Proceedings of the IEEE*, **95**, 467-506. http://dx.doi.org/10.1109/JPROC.2006.890107

[6] Danowitz, A., *et al*. (2012) CPU DB: Recording Microprocessor History. Communications ACM, USA.

[7] Patterson, D. and Hennessy, J. (2014) Computer Organization and Design: The Hardware/Software Interface. 5th Edition, Morgan Kaufmann, Burlington.

[8] Ostrovsky. I. (2010) Gallery of Processor Cache Effects. http://igoro.com/archive/gallery-of-processor-cache-effects

[9] Meyers, S. (2011) CPU Caches and Why You Care. http://www.aristeia.com/TalkNotes/ACCU2011_CPUCaches.pdf

[10] Meijer, P. (2011) Tearing Systems of Differential Algebraic Equations. Master's Thesis, Lund University, Sweden.

[11] Cellier, F.E. and Kofman, E. (2006) Continuous System Simulation. Springer, Berlin, Chapter 2, 25-56.

[12] Cellier, F.E. and Kofman, E. (2006) Continuous System Simulation. Springer, Berlin, Chapter 8, 319-396.

[13] Lambert, J.D. (1991) Numerical Methods for Ordinary Differential Systems: The Initial Value Problem. John Wiley & Sons, Inc., Hoboken, Chapter 2, 21-44.

[14] Fehlberg, E. (1968) Classical Fifth-, Sixth-, Seventh-, and Eighth-Order Runge-Kutta Formulas. Technical Report TR R-287, NASA Johnson Space Center, Houston.

[15] Elmqvist, H., Otter, M. and Cellier, F.E. (1995) Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential-Algebraic Equation Systems. *Proceedings of ESM* 95, *European Simulation Multiconference*. https://www.inf.ethz.ch/personal/cellier/Pubs/OO/esm_95.pdf

[16] Flynn, M. (1972) Some Computer Organizations and Their Effectiveness *IEEE Transactions on Computers*, **C-21**, 948-960.

[17] Rauber, T. and Rünger, G. (2013) Parallel Programming for Multicore and Cluster Systems. Springer, Berlin.

[18] Hadoop. http://hadoop.apache.org/

[19] Intel. http://www.intel.com

[20] AMD. http://www.amd.com

[21] Patterson, D. and Hennessy, J. (2014) Computer Organization and Design: The Hardware/Software Interface. 5th Edition, Morgan Kaufmann, Burlington, Chapter 6, 500-575.

[22] Patterson, D. and Hennessy, J. (2014) Computer Organization and Design: The Hardware/Software Interface. 5th Edition, Morgan Kaufmann, Burlington, Chapter 5, 372-499.

[23] GNU Scientific Library. http://www.gnu.org/software/gsl/

[24] Foster, I. (1995) Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. Addison-Wesley, Boston, Chapter 2, 27-82.

[25] Modelica. https://www.modelica.org/

[26] Dassault Systems AB (2013) Dymola User Manual Volume 1.

[27] GCC. The GNU Compiler Collection. https://gcc.gnu.org/

[28] McSherry, F., Isard, M. and Murray, D.G. (2015) Scalability! But at What COST? 15*th Workshop on Hot Topics in Operating Systems* (*HotOS XV*), Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry

[29] Anderson, T.E. (1990) The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, **1**, 6-16. http://dx.doi.org/10.1109/71.80120

[30] The C++ Programming Language. https://isocpp.org/

[31] Gear, C.W. and Wells, D.R. (1984) Multirate Linear Multistep Methods. *BIT Numerical Mathematics*, **24**, 484-502. http://dx.doi.org/10.1007/BF01934907

[32] Howe, R.M. (1998) Real-Time Multirate Synchronous Simulation with Single and Multiple Processors. *Proceedings SPIE Proceedings*, **3369**, Enabling Technology for Simulation Science II. http://dx.doi.org/10.1117/12.319349

[33] Dassault Systems AB (2013) Dymola User Manual Volume 2.

[34] Elmqvist, H., Mattsson, S.E. and Olsson, H. (2002) New Methods for Hardware-in-the-Loop Simulation of Stiff Models. *Proceedings of the* 2*nd International Modelica Conference*, Oberpfaffenhofen, 18-19 March 2002, 59-64.

[35] Sundials. http://computation.llnl.gov/casc/sundials/main.html

[36] Lawrence Livermore National Laboratory. https://www.llnl.gov/

[37] Cellier, F.E. and Kofman, E. (2006) Continuous System Simulation. Springer, Berlin, Chapter 3, 57-116.